

The JSOC and SHA Stanford Data Providers

E. J. Mansky

1 Introduction

The Joint Science Operations Center (JSOC) at Stanford stores data for the Atmospheric Imaging Assembly (AIA) and Helioseismic and Magnetic Imager (HMI) instruments on board the Solar Dynamic Observatory (SDO) spacecraft, as well as the Stanford Helioseismology Archive (SHA) of the Solar and Heliospheric Observatory (SOHO) spacecraft's Michelson Doppler Imager (MDI) instrument. The data from these instruments are currently highly popular data in the solar community. The data are stored using the JSOC's Data Record Management System (DRMS) and Storage Unit Management System (SUMS) software packages, which use Postgres databases and are largely written in C (although some movement towards python has recently taken place). Allowing these data to be searched for in the Virtual Solar Observatory (VSO) system developed outside of Stanford is thus a matter of developing VSO "data provider" software to allow the VSO software to interface with the DRMS/SUMS software. This document was developed as a result of adding the MDI data to the set of data that is searchable by the VSO by installing an MDI data provider at the JSOC, similar to the already existing AIA and HMI data providers installed at remote sites. The intent is to document the existing data providers, and possibly facilitate future additions.

The vast majority of Data Providers have the Flexible Image Transport System (FITS) files containing their data residing on disc at their sites, and the files themselves are fairly flat in that a simple search of a date range, wavelength and/or other physical observables will be sufficient to retrieve data from the DB. The JSOC and SHA data differ in three fundamental ways: (a) a complete FITS files, including header meta-data, must be constructed on-the-fly for each query request since no FITS files are stored directly at the JSOC since they intentionally lack meta-data which is stored instead in DRMS ; (b) the precision of the time data of the instruments is such that special conversion between the dynamic time the instruments record and the UNIX time the VSO will be using for time-based searches is required due to the omission of leapseconds in both time variables; and (c) the image meta-data itself is saved in multiple tables that are organized in series and are roughly equivalent to physical observables. Examples of multiple series include the storage of vector magnetograms, polarization data and Doppler data. The individual series themselves are not directly

searchable or selectable by the end User.

The result of these 3 major differences is that the VSO codebase needs extensions and modifications to handle the more complex requirements of JSOC and SHA data. We describe here in detail the modifications and extensions that are used for the JSOC and SHA data.

Figure 1 illustrates the overall flow of a query through the VSO codebase to retrieve JSOC and SHA data, and return that data as complete FITS files.

The VSO codebase first reads all the registry .xml files in the REG_DIR configuration folder at the start of a search. These registry files define several key variables for each Data Provider including the `<proxy>` and `<uri>` tags which define the endpoint URL where the data resides, and the namespace wherein the methods called by the VSO codebase should reside (`Query` and `GetData` being the primary ones).

Once the end User completes the search query and begins their search, the SOAP code packages up the query and sends it off to the various endpoint URLs specified by their respective `<proxy>` tag values in their registry files. The JSOC and SHA code resides in separate namespaces, hence they have separate endpoint CGI scripts, `Stanfordi` and `SHAi`, respectively.

In the `Stanfordi` and `SHAi` CGI scripts are package statements that define their respective namespaces, specified in the `<uri>` tag, `VSO::JSOCi` and `VSO::SHAi`, respectively.

The VSO codebase then begins the query for data by calling, one per thread, the Data Providers that have been selected in the User query. The general `DataProvider` object is first initialized and then either the `JSOC` object or the `SHA` object are created to complete the initialization of the respective `DataProvider` objects. As part of the initialization, the `JSOC` and `SHA` objects read all their respective series registry files that define the meta-data such as DRMS table names, shadow table names, DRMS series names, and various regular expressions (RE) that are used to construct the argument list that is passed to the final CGI script in the process, the DRMS Export CGI script, `drms-export.cgi` and `drms-export-sha.cgi` for JSOC and SHA, respectively. These export CGI scripts in turn call the DRMS C code `drms-export-to-stdout` that does the actual FITS file construction and tar-ing (if chosen).

The `JSOC` and `SHA` `DataProvider` objects use a number of public and private methods (detailed below) to process the records returned in the query from the DRMS DB, and construct links on the Cart Request Status page to the DRMS export CGI, one link per series, for the final data selected by the User on the main search results page.

In §2 we cover in detail the `DataProvider` objects and associated methods for `JSOC`, while in §3 we cover the corresponding code for `SHA`. §4 enumerates the shadow tables that are used and the triggers that populate them. The key field `fileid` in the shadow tables is covered in detail due to its central role in the construction of the argument list for the DRMS Export CGI scripts. The series in `JSOC` and `SHA` are examined in §5. We close by examining in §6 the steps needed to create a new `JSOC/SHA` type of `DataProvider`.

Topics we do not discuss here include the JMD, slony replication, the DB

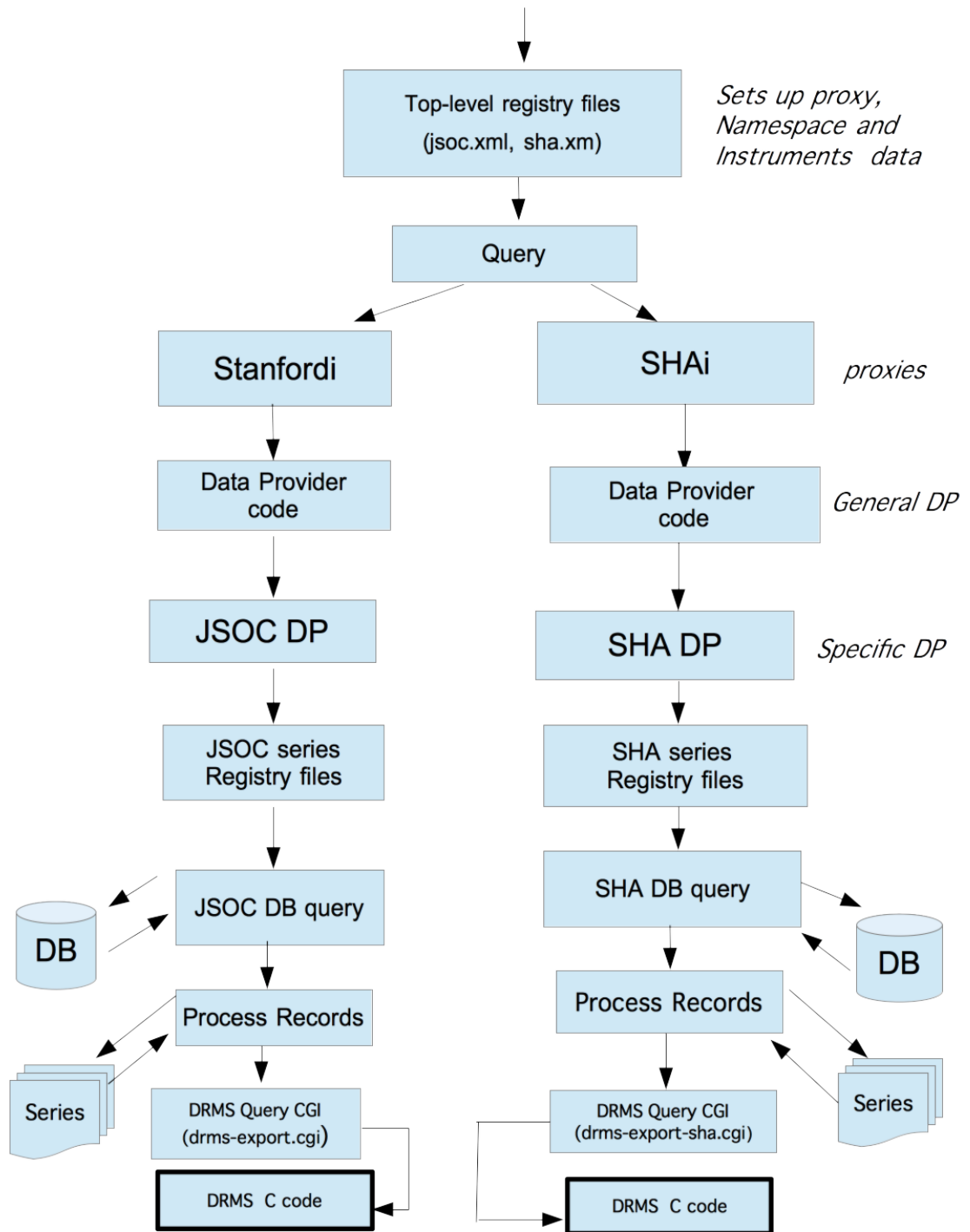


Fig. 1: Overall VSO code flow for JSOC and SHA queries

schema and the DRMS system itself. These latter topics are covered by other wiki entries.

2 JSOC

The file JSOC.pm contains three namespaces, `DataProvider::JSOC`, `DataProvider::JSOC::Query`, `DataProvider::JSOC::Series`, each with their own public and private methods.

1) `DataProvider::JSOC` *base is DataProvider::SQL*

Description: Main namespace for JSOC, and initial entry point from Core into the specific JSOC code.

Public Methods:

`Registry` returns a new `JSOC::Registry` object

`Config` returns a new global `Config` object

Private Methods:

`_ProviderID` returns string `JSOC`

`_GetQueryType` returns namespace of `Query` type, for `JSOC` that is:
`Physics::Solar::VSO::JSOC::Query`

`_GetDatabaseInfo` returns array of data needed to establish a
DB connection

`_SupportedDataMethods` returns a hash of supported `getdata_method`
values

`_ProcessDataRequest` main routine that handles the incoming
data request from the Core

`__GetSeriesObject` called by `_ProcessDataRequest`
note the two underscores

2) `DataProvider::JSOC::Query` *base is DataProvider::Query*

Description: JSOC-specific namespace that contains `Query`-related methods. Parent is the main `DataProvider::Query` namespace.

Public Methods:

`GetDatabase` returns `dataprovder` Object's `GetDatabase`, `dataprovder`
is a `Public` method in parent namespace `DataProvider::Query`

Helper functions: `limit`, `params`

Private Methods:

`_GetSeriesMatches` matches the series list passed in with the series
generated from the `Registry`

`_GetSeriesObjects` returns an array of `Objects` after calling `_GetSeriesObject`
in a loop over all the series that match

`__GetSeriesObject` creates and initializes `Objects` in the `Series` namespace
note the two underscores

`_Count` sums the return value of `_Count` from each of the Series Objects and returns the total

`_SendSearch` sends the search query to each series Object and the results are the return value of `_PackageMultipleResults`

Note special handling of 3 cases: query includes params *near* or *sample*, or the count of returned values exceeds the value of limit

`_SendSearch_Summary` called only in the case where count \geq limit in `_SendSearch`

`_SendSearch_Near` called only in the case where the params *near* is specified in the query

`_PackageMultipleResults` returns the argument list either unchanged, or cast as an array if the no. of elements > 1

3) `DataProvider::JSOC::Series` *base is* `DataProvider::Query::SQL`
Description: JSOC-specific namespace for JSOC Series. Parent is main `DataProvider::Query::SQL` namespace. Acts as the grandparent class for all the AIA and HMI series. See Figures 3 and 4 for details.

Public Methods:

`init` initializes Object by calling parent `init` and this namespace's `ProcessQueryParams`. The query itself is stored in Object attribute *queryobj*

`QueryObj` returns the Object's attribute *queryobj* which is just the query being processed.

`GetDatabase` returns DB handle

`ProcessQueryParams` validates the parameters in the query by calling two types of private functions: `_CheckValid_X` and `_ProcessParam_X` in each series

namespace if they exist (X being the series names). NOTE: This method is only run once at the beginning of the search process.

Helper function: `debug`

Helper function: `get_download.cgi` key method that returns the DRMS query export CGI URL. Uses `DataProvider::JSOC::Sites` to determine the URL, separate from

`webui_config.xml` or the value of `< proxy >` in the top-level JSOC registry file (`jsoc.xml`)

Helper function: `format_date` matches the input `$vso_date` to a 7-field RE and returns that date re-formatted, otherwise returns an empty string

Helper function: `vso_date` lots of date manipulation of input date string, outputs reformatted date if RE is matched, otherwise an empty string

Helper function: `isLeapYear` determines if current year in query is a leap year

Private Methods:

`_ProcessRecord` defines a closure call of the private series methods `_ProcessRecord_X` where X is one of 10 enumerated fields (hard-coded)

+ any additional fields in the query itself

`_ProcessCluster` defines a closure call of the private series methods `_ProcessRecord_X` where X is one of 11 enumerated fields (hard-coded) + any additional fields in the query itself. Only called for Summary row processing (ie. when count > limit). NOTE: The extra field is *pptid*

`_MakeProcess` a factory that sets up a function `_ProcessParam_X` where X is the param passed in the argument list. Each function defined makes use of a mapping table or not, depending on the caller, to add a piece of SQL to the DB query that is being constructed. Overrides parent method in `DataProvider`. *note the two underscores in the name*

`_BuildWhereString` add an AND clause to the *where_string* Object attribute

`_Count` uses `GetDatabase` and `_BuildWhereString` to prepare and execute the count SQL. Returns the count.

`_SendSearch_Near` if series method `sql_near` is defined, then the SQL in it is prepared and executed and the returning records processed by `_ProcessRecord` and then returned via `_PackageResults`, the main method used to return all found data to the parent `DataProvider` code. Currently only the AIA series has `sql_near`. Otherwise a call the `_SendSearch` is returned.

`_SendSearch_Sampled` similar to `_SendSearch_Near` except for series method `sql_sampled`. Currently both AIA and HMI series support `sql_sampled`

`_SendSearch_Summary` similar to `_SendSearch_Summary` except for series methods `sql_cluster_count` and `sql_cluster`. Currently both AIA and HMI support cluster SQL statements.

`_SendSearch` main routine to prepare and execute the SQL and return the results by sending them through `_ProcessRecord` and finally returned via `_PackageResults`.

NOTE: If series method `sql_cluster` is defined and count > limit, then only the result of `_SendSearch_Summary` is returned.

Helper function: `_PackageData` calls `dataprovider` Object method `_PackageData`

Helper function: `_ThrowGetDataError` calls `dataprovider` Object method `_ThrowGetDataError`

Figure 2 illustrates in a little more detail the flow in the VSO code in the JSOC namespace.

At the top of the figure, we assume the User's query has been returned, via Query, and that the main `GetData` function call is now occurring. The resultant call in the core to `GetData` yields a call to method `_ProcessDataRequest` in `DataProvider2` which gets passed to the `DataProvider::JSOC` method of the same name via inheritance, which is shown as a dotted line. Once inside `DataProvider::JSOC::_ProcessDataRequest`, the key data hash `$fileids{$series}` is defined and a loop over all elements of `$fileids` is done. In the loop a match is first made between all the series in the results data and the series defined for JSOC, with subsequent tests made to determine if any errors were

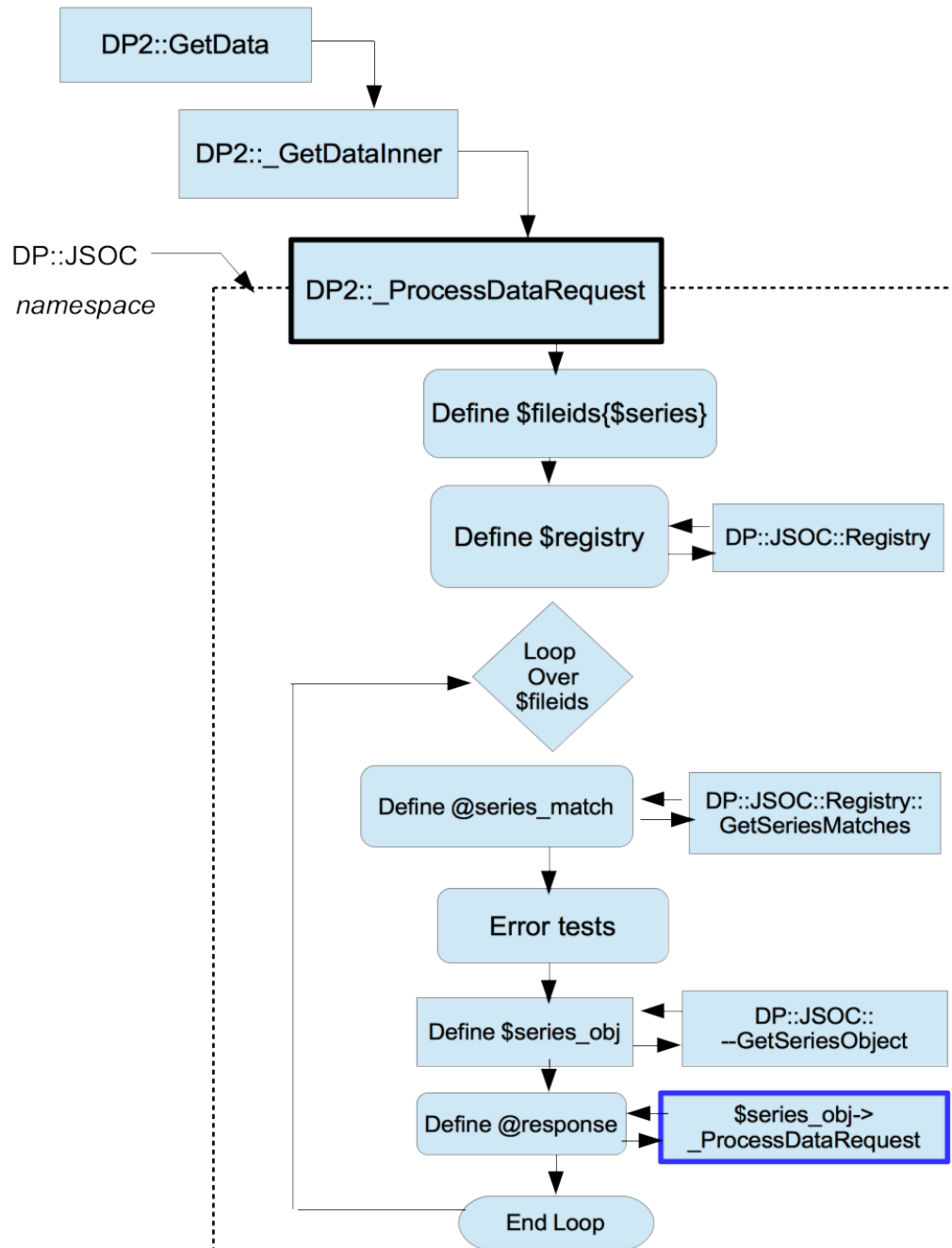


Fig. 2: Details of the code flow in JSOC

encountered. If no errors were encountered, then a call to the private function `_GetSeriesObject` is done and then a call, in turn, to each series object's private method `_ProcessDataRequest` is made to get the actual data for that series. The results are then pushed into the main data structure, `@response` for further processing.

2.1 Related Config Variables

The variables in `Config.xml` that are used solely by JSOC are:

<code>JSOC_EXPORT_CGI</code>	Full URL to CGI script used for DRMS exports by JSOC
<code>JSOC_EXPORT_CMD</code>	Full path to C code file used by <code>JSOC_EXPORT_CGI</code> for JSOC exports
<code>JSOC_EXPORT_CMD_TYPE</code>	Argument passed to C code in <code>JSOC_EXPORT_CMD</code>
<code>REG_DIR_JSOC</code>	Path to store JSOC series Registry files
<code>REG_DUMP_JSOC</code>	Path to store pre-parsed JSOC series registry data

3 SHA

The SHA code is nearly identical to that used by JSOC. One difference that should be noted is that there is no equivalent of `Sites.pm` in the SHA code. Hence the MDI version of `get_download.cgi` returns `undef`, so that the config variable `SHA_EXPORT_CGI` is used as the primary means of pointing the MDI code to the DRMS export CGI scriptname.

3.1 Related Config Variables

The variables in `Config.xml` that are used solely by SHA are:

<code>SHA_EXPORT_CGI</code>	Full URL to CGI script used for DRMS exports by SHA
<code>SHA_EXPORT_CMD</code>	Full path to C code file used by <code>SHA_EXPORT_CGI</code> for SHA exports
<code>SHA_EXPORT_CMD_TYPE</code>	Argument passed to C code in <code>SHA_EXPORT_CMD</code>
<code>REG_DIR_SHA</code>	Path to store SHA series Registry files
<code>REG_DUMP_SHA</code>	Path to store pre-parsed SHA series registry data

4 Shadow Tables and Triggers

One reason shadow tables are used in both the JSOC and SHA `DataProvider` code is because the dynamic times in the original data tables are not directly compatible with the UNIX times formulated in a VSO search. Databases provide at least two mechanisms to solve the problem of incompatible data. One is through the use of views, and the other is by using auxiliary, or shadow tables.

For both JSOC and SHA, shadow tables were chosen. The shadow tables contain both the time data that is compatible with the UNIX search data, and a constructed field, `fileid`, composed of the series name concatenated with the

`t_rec_index` value for that row. The constructed field `fileid` is used to arrange an ordered list of series names with their respective `t_rec_index` values so that the CGI argument `record` to the DRMS Query export script can be sorted by series name.

For both JSOC and SHA, the data resides in a PostgreSQL database. Each instrument is kept in it's own namespace. For AIA the namespace is `aia` and HMI uses `hmi`. The shadow tables are in namespace `vso`. At SDAC and NSO the corresponding triggers are installed in namespace `public`, while at Stanford they are installed in namespace `vso`.

Table 1 summarizes the original and shadow tables for AIA, together with their respective triggers.

Tab. 1: AIA Original and Shadow Tables with their Triggers

Original Table	Shadow Table	Trigger name	Comments
<code>aia.lev1</code>	<code>vso.aia_lev1</code>	<code>proc_update_vso_aia_lev1</code>	<code>fileid = 3 fields</code>

For shadow table `vso.aia_lev1` the 3 fields comprising `fileid` are:

```
shadow_table_name : wave : t_rec_index
```

Table 2 summarizes the equivalent data for HMI.

Tab. 2: HMI Original and Shadow Tables with their Triggers

Original Table	Shadow Table	Trigger name	Comments
<code>hmi.Ic_45s</code>	<code>vso.hmi_Ic_45s</code>	<code>proc_update_vso_hmi_Ic_45s</code>	<code>fileid = 3 fields</code>
<code>hmi.Ic_720s</code>	<code>vso.hmi_Ic_720s</code>	<code>proc_update_vso_hmi_Ic_720s</code>	<code>fileid = 3 fields</code>
<code>hmi.Ld_45s</code>	<code>vso.hmi_Ld_45s</code>	<code>proc_update_vso_hmi_Ld_45s</code>	<code>fileid = 3 fields</code>
<code>hmi.Ld_720s</code>	<code>vso.hmi_Ld_720s</code>	<code>proc_update_vso_hmi_Ld_720s</code>	<code>fileid = 3 fields</code>
<code>hmi.Lw_45s</code>	<code>vso.hmi_Lw_45s</code>	<code>proc_update_vso_hmi_Lw_45s</code>	<code>fileid = 3 fields</code>
<code>hmi.Lw_720s</code>	<code>vso.hmi_Lw_720s</code>	<code>proc_update_vso_hmi_Lw_720s</code>	<code>fileid = 3 fields</code>
<code>hmi.M_45s</code>	<code>vso.hmi_M_45s</code>	<code>proc_update_vso_hmi_M_45s</code>	<code>fileid = 3 fields</code>
<code>hmi.M_720s</code>	<code>vso.hmi_M_720s</code>	<code>proc_update_vso_hmi_M_720s</code>	<code>fileid = 3 fields</code>
<code>hmi.S_720s</code>	<code>vso.hmi_S_720s</code>	<code>proc_update_vso_hmi_S_720s</code>	<code>fileid = 3 fields</code>
<code>hmi.V_45s</code>	<code>vso.hmi_V_45s</code>	<code>proc_update_vso_hmi_V_45s</code>	<code>fileid = 3 fields</code>
<code>hmi.V_720s</code>	<code>vso.hmi_V_720s</code>	<code>proc_update_vso_hmi_V_720s</code>	<code>fileid = 3 fields</code>

There are additional HMI series in the DB that are currently not being served via the VSO codebase, including multiple B and vw series.

For the HMI shadow tables , the 3 fields comprising `fileid` are:

```
shadow_table_name : t_rec_index : t_rec_index
```

Table 3 summarizes the equivalent data for MDI.

Tab. 3: MDI Original and Shadow Tables with their Triggers

Original Table	Shadow Table	Trigger name	Comments
<i>mdi</i> .fd_I0_extract	<i>vso</i> .mdi__fd_I0_extract	proc_update_vso_mdi__i0_extract	<i>fileid</i> = 2 fields
<i>mdi</i> .fd_I0	<i>vso</i> .mdi__fd_I0	proc_update_vso_mdi__i0	<i>fileid</i> = 2 fields
<i>mdi</i> .fd_lc	<i>vso</i> .mdi__fd_Ic	proc_update_vso_mdi__ic	<i>fileid</i> = 2 fields
<i>mdi</i> .fd_Ld	<i>vso</i> .mdi__fd_Ld	proc_update_vso_mdi__ld	<i>fileid</i> = 2 fields
<i>mdi</i> .fd_M_96m_lev182	<i>vso</i> .mdi__fd_M_96m_lev182	proc_update_vso_mdi__m_96m_lev182	<i>fileid</i> = 2 fields
<i>mdi</i> .fd_M_extract	<i>vso</i> .mdi__fd_M_extract	proc_update_vso_mdi__m_extract	<i>fileid</i> = 2 fields
<i>mdi</i> .fd_M_lev182	<i>vso</i> .mdi__fd_M_lev182	proc_update_vso_mdi__m_lev182	<i>fileid</i> = 2 fields
<i>mdi</i> .fd_V_bin2x2_30s	<i>vso</i> .mdi__fd_V_bin2x2_30s	proc_update_vso_mdi__v_bin2x2_30s	<i>fileid</i> = 2 fields
<i>mdi</i> .fd_V_bin2x2	<i>vso</i> .mdi__fd_V_bin2x2	proc_update_vso_mdi__v_bin2x2	<i>fileid</i> = 2 fields
<i>mdi</i> .fd_V_extract	<i>vso</i> .mdi__fd_V_extract	proc_update_vso_mdi__v_extract	<i>fileid</i> = 2 fields
<i>mdi</i> .fd_V	<i>vso</i> .mdi__fd_V	proc_update_vso_mdi__v	<i>fileid</i> = 2 fields
<i>mdi</i> .hr_I0	<i>vso</i> .mdi__hr_I0	proc_update_vso_mdi__i0	<i>fileid</i> = 2 fields
<i>mdi</i> .hr_lc	<i>vso</i> .mdi__hr_Ic	proc_update_vso_mdi__lc	<i>fileid</i> = 2 fields
<i>mdi</i> .Ld_bin2x2	<i>vso</i> .mdi__hr_Ld_bin2x2	proc_update_vso_mdi__ld_bin2x2	<i>fileid</i> = 2 fields
<i>mdi</i> .hr_Ld	<i>vso</i> .mdi__hr_Ld	proc_update_vso_mdi__ld	<i>fileid</i> = 2 fields
<i>mdi</i> .hr_M_bin2x2	<i>vso</i> .mdi__hr_M_bin2x2	proc_update_vso_mdi__m_bin2x2	<i>fileid</i> = 2 fields
<i>mdi</i> .hr_M	<i>vso</i> .mdi__hr_M	proc_update_vso_mdi__m	<i>fileid</i> = 2 fields
<i>md</i> .hr_V_12s	<i>vso</i> .mdi__hr_V_12s	proc_update_vso_mdi__v_12s	<i>fileid</i> = 2 fields
<i>mdi</i> .hr_V_bin2x2	<i>vso</i> .mdi__hr_V_bin2x2	proc_update_vso_mdi__v_bin2x2	<i>fileid</i> = 2 fields
<i>mdi</i> .hr_V	<i>vso</i> .mdi__hr_V	proc_update_vso_mdi__v	<i>fileid</i> = 2 fields
<i>mdi</i> .rwbin_lc	<i>vso</i> .mdi__rwbin_Ic	proc_update_vso_mdi__rwbin_lc	<i>fileid</i> = 2 fields
<i>mdi</i> .rwbin_Ld	<i>vso</i> .mdi__rwbin_Ld	proc_update_vso_mdi__rwbin_ld	<i>fileid</i> = 2 fields
<i>mdi</i> .vw_V	<i>vso</i> .mdi__vw_V	proc_update_vso_mdi__vw_v	<i>fileid</i> = 2 fields

For the MDI shadow tables , the 2 fields comprising `fileid` are:

```
shadow_table_name : t_rec_index
```

4.1 Scripts for creating and installing shadow tables and triggers

The scripts to create and install the shadow tables and triggers for AIA and HMI are:

```
create_vso_trigger.sql
shadow_aia_template.pl
shadow_hmi_template.pl
```

which are located in the CVS tree in `DataProviders/JSOC` and `DataProviders/JSOC/db_triggers`, respectively.

The corresponding script for MDI is `create_mdi_shadow_tables.sql` in `DataProviders/SHA`.

4.2 Dynamic Time Conversion

The conversion of the dynamic time recorded by the AIA, HMI and MDI instruments, and the UNIX time is done via the SQL function `dynamic_to_unix`. The SQL code for the function is stored in the file `create_vso_trigger.sql`.

UNIX time is the number of seconds that have elapsed since January 1, 1970 midnight UTC time, *omitting* leap seconds. The dynamical time stored in the PostgreSQL DB also omits leap seconds, hence the need for a conversion to correctly account for them.

Code snippet from `dynamic_to_unix`:

```
BEGIN
CASE
    WHEN dynamical >= 1009843234.0 THEN
        leapseconds := 19.0;
    WHEN dynamical >= 915148833.0 THEN
        ... CASE statement defining leapseconds for different years

    ELSE
        leapseconds := -15.0;
END CASE;

— difference between dynamical and internal time found via
— set timezone to utc;
— select extract(epoch from timestamp '1976-12-31 23:59:45');
— as of postgres 8.4 is 220924785 seconds

utc_time = to_timestamp( dynamical - leapseconds + 220924785);

RETURN utc_time;
```

5 Series

The JSOC and SHA data pipelines break down the storage of their data into multiple series of tables for a given type of physical observable. The overall intent is to allow the SDO data from AIA & HMI, and the SOHO MDI, data to be stored in the DRMS system and hide the existence of the series structure from direct query via the VSO thereby (hopefully) making the searching for data simpler. The assumption being that the User will want all available physical observables for a given instrument and date range

Each series has its own package file defining the namespace wherein the public methods needed to fully define that series are located.

Figure 3 expands upon Figure 2 by further detailing the chain of calls involved in `_ProcessDataRequest` for each series (the blue rectangle at the lower right of the figure). The downward pointing solid black arrow from the box with `$series_obj->_ProcessDataRequest` to `DP::JSOC::AIA::aia__lev1` represents a direct call to the latter namespace. The base class for `DP::JSOC::Series::aia__lev1` is `DP::JSOC::Series::AIA` (represented by a blue dashed rectangle in Fig. 3). Since the called method `_ProcessDataRequest` does *not* exist in the `DP::JSOC::Series::aia__lev1` namespace, the parent is checked to see if the method exists there (represented by the green dashed upward arrow). Once resolved, control is returned to the original caller (shown as an upward pointing solid black arrow).

The bottom part of Fig. 3 shows, in outline form, the call chain involved in processing the parameters in the User's query. The chain starts from `DP::Query` which calls the private method `DP::_Search` with the argument `$query` representing the User's query as a hash. The method `_Search` then calls the private method `_GetQueryObject` which obtains the class of the series `Query` from `DP::JSOC::_GetQueryType`. The value of `$class` returned by `DP::JSOC::_GetQueryType` becomes the Object `$obj`. A loop over the parameters in the User query is executed next in order to verify each parameter's value by calling the private method `_ProcessParam_param` in each series namespace. The one parameter that is explicitly in `DP::JSOC::Series::aia__lev1` is *dark* (i.e. `_ProcessParam_dark`), the call to which is represented by a thin solid black arrow pointing downward in the lower right of Fig. 3. The text *... other params* represents the calls to the other parameters in the User query. For AIA, the remainder of these calls are resolved in the parent namespace, `DP::JSOC::Series::AIA`, represented by the fine dotted upward green arrow in Fig. 3. Control is returned to the loop from the parent namespace, represented by the dotted purple downward arrow, wherein processing continues.

5.1 JSOC

5.1.1 AIA

The series for AIA are summarized in Tables 4 -6.

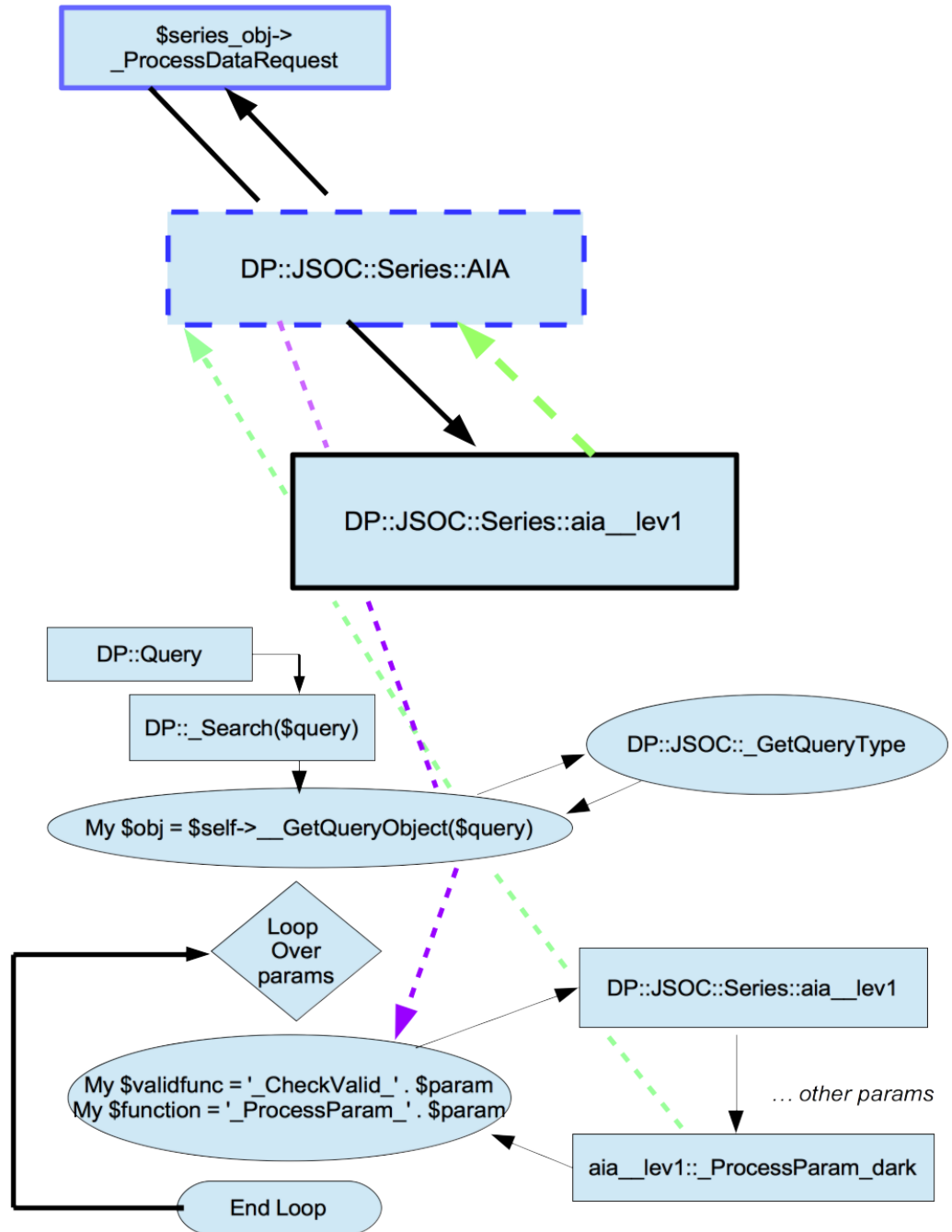


Fig. 3: Inheritance Diagram for AIA series

Tab. 4: AIA Series

Series Name	Shadow Table Name	Physical Observable	Parent Namespace
<i>aia</i> .lev1	<i>vso</i> . <i>aia</i> _lev1		AIA
<i>aia</i> . <i>test</i> .synoptic2	synoptic		

Tab. 5: AIA Series Parameters and Records

Series Name	Parameter (_ProcessParam_X)	Record _ProcessRecord_X	sql_X
<i>aia</i> .lev1	dark	series	prefix
		info	suffix
		level	near
		fileid	sampled
		size	cluster
		exptime	
		eclipse	
		dark	
		precentd	
		sci_mode	

Tab. 6: AIA Parent Series

Parent Series	Parameter (_ProcessParam_X)	Record _ProcessRecord_X	sql_X
AIA	extent	extent	count
	time	time	prefix
	wave	wave	sampled
	fileid	fileid	suffix
	pptid	pptid	cluster
			cluster_suffix
			cluster_count

5.1.2 HMI

The series for HMI are summarized in Tables 7-9.

Tab. 7: HMI Series

Series Name	Info field	Cadence	Parent Namespace
hmi__Ic_45s	Continuum Intensity	45s	HMI
hmi__Ic_720s	Continuum Intensity	720s	HMI_720s
hmi__Ld_45s	Line depth	45s	HMI_720s
hmi__Ld_720s	Line depth	720s	HMI_720s
hmi__Lw_45s	Line width	45s	HMI_720s
hmi__Lw_720s	Line width	720s	HMI_720s
hmi__M_45s	Magnetogram	45s	HMI
hmi__M_720s	Magnetogram	720s	HMI_720s
hmi__S_720s	Stokes parameters (I, Q, U, V)	720s	HMI HMI
hmi__V_45s	Dopplergram	45s	HMI
hmi__V_720s	Dopplergram	720s	HMI_720s

Tab. 8: HMI Series Parameters and Records

Series Name	Parameter _ProcessParam_X	Record _ProcessRecord_X
hmi__Ic_45s		info, series, fileid
hmi__Ic_720s		info, series, fileid
hmi__Ld_45s		info, series, fileid
hmi__Ld_720s		info, series, fileid
hmi__Lw_45s		info, series, fileid
hmi__Lw_720s		info, series, fileid
hmi__M_45s		info, series, fileid
hmi__M_720s		info, series, fileid
hmi__S_720s	info, series, fileid	
hmi__V_45s	info, series, fileid	
hmi__V_720s	info, series, fileid	

5.2 SHA

5.2.1 MDI

There are 2 types of general series for MDI: Full Disk and High Resolution, and 2 types of weighted series: Radially and vector weighted. The complete set of series for MDI are summarized in Tables 10-12.

Tab. 9: HMI Parent Series

Parent Series	Parameter _ProcessParam_X	Record _ProcessRecord_X	sql_X
HMI	extent	extent	count
		wave	prefix
	time	time	sampled
	fileid	fileid	suffix
	pptid	pptid	cluster
			cluster_suffix
			cluster_count

Tab. 10: MDI Series

Series Name	Info Field	Cadence	Comments
mdi__fd_I0.extract	Full Disk Intensity		extracted subfield
mdi__fd_I0	Full Disk Intensity		
mdi__fd_Ic	Full Disk Intensity		Doppler corrected
mdi__fd_Ld	Full Disk Line Depth		
mdi__fd_M_96m_lev182	Full Disk Magnetic field	96min	Level 1.8
mdi__fd_M.extract	Full Disk Magnetic field		extracted subfield
mdi__fd_M_lev182	Full Disk Magnetic field		Level 1.8
mdi__fd_V_bin2x2_30s	Full Disk Doppler	30sec	binned 2X2
mdi__fd_V_bin2x2	Full Disk Doppler		binned 2X2
mdi__fd_V.extract	Full Disk Doppler		extracted subfield
mdi__fd_V	Full Disk Doppler		
mdi__hr_I0	High Resolution Intensity		
mdi__hr_Ic	High Resolution Intensity		Doppler corrected
mdi__hr_Ld_bin2x2	High Resolution Line Depth		binned 2X2
mdi__hr_Ld	High Resolution Line Depth		
mdi__hr_M_bin2x2	High Resolution Magnetic Field		binned 2X2
mdi__hr_M	High Resolution Magnetic Field		
mdi__hr_V_12s	High Resolution Doppler	12sec	
mdi__hr_V_bin2x2	High Resolution Doppler		binned 2X2
mdi__hr_V	High Resolution Doppler		
mdi__rwbin_Ic	Radially weighted Intensity		Doppler corrected and binned 8X8
mdi__rwbin_Ld	Radially weighted Line Depth		binned 8X8
mdi__vw_V	Vector weighted Doppler		

Tab. 11: MDI Series Parameters and Records

Series Name	Record _ProcessRecord_X	Parent Series
mdi__fd_I0.extract	info, series, fileid	MDI
mdi__fd_I0	info, series, fileid	MDI
mdi__fd_Ic	info, series, fileid	MDI
mdi__fd_Ld	info, series	MDI
mdi__fd_M_96m_lev182	info, series, fileid	MDI
mdi__fd_M.extract	info, series, fileid	MDI
mdi__fd_M_lev182	info, series, fileid	MDI
mdi__fd_V_bin2x2_30s	info, series, fileid	MDI
mdi__fd_V_bin2x2	info, series, fileid	MDI
mdi__fd_V_extract	info, series, fileid	MDI
mdi__fd_V	info, series, fileid	MDI
mdi__hr_I0	info, series, fileid	MDI
mdi__hr_Ic	info, series, fileid	MDI
mdi__hr_Ld_bin2x2	info, series, fileid	MDI
mdi__hr_Ld	info, series, fileid	MDI
mdi__hr_M_bin2x2	info, series, fileid	MDI
mdi__hr_V_12s	info, series, fileid	MDI
mdi__hr_V_bin2x2	info, series, fileid	MDI
mdi__hr_V	info, series, fileid	MDI
mdi__rwbin_Ic	info, series, fileid	MDI
mdi__rwbin_Ld	info, series, fileid	MDI
mdi__vw_V	info, series, fileid	MDI

Tab. 12: MDI Parent Series

Parent Series	Parameter _ProcessParam_X	Record _ProcessRecord_X	sql_X
MDI	extent	extent	count
		wave	prefix
	time	time	sampled
	fileid	fileid	suffix
	pptid	pptid	cluster
			cluster_suffix
			cluster_count

6 Creating new JSOC/SHA type Data Providers

To add a new type of Data Provider that follows the mechanisms used by JSOC and SHA, do the follow:

- 1) Add new scripts to create all shadow tables and triggers. Run scripts to install the tables and triggers into the DB.
- 2) Add new script to create all series namespaces. Run script to create all new series namespace .pm files required.
- 3) Create top-level Registry file and corresponding CGI script that the `<proxy>` tag points to.
- 4) Create series registry .xml files to store the meta-data of each series in the new Data Provider.
- 5) Create new .pm files similar to JSOC.pm, SHA.pm, AIA.pm, HMI.pm and MDI.pm to store all needed public and private methods for the new namespaces required. Depending on the requirements of the new Data Provider code, this step may be completely straightforward, or require a lot of new code and methods to achieve.
- 6) Complete the export process of FITS files Depending on whether the FITS already exist, or need to be constructed, on-the-fly, the export process may be similar to that used by JSOC and SHA, or much simpler.