JSOC Dataset naming convention for use with the DRMS

version 6.3, 14 December 2007

Data is stored in the HMI/AIA JSOC in many "Data Series." A Data Series (or dataseries) is a basic sequence of like data objects, typically "images" or other binary data along with associated meta-data. A dataseries consists of a sequence of Data Records. Usually, each datarecord is the data for one step in "time". Most but certainly not all dataseries are sequences in time. They can be in principle any list of data objects. A good way to think about a dataseries is as a table of rows and columns where the columns contain meta-data or data arrays and the rows (i.e. records) contain all the meta-data and data arrays for an instance in time.

The actual storage methods for the meta-data and array data does not matter for the description of the naming conventions but some details will help in understanding the use of the naming system.

A datarecord consists of Keyword tagged meta-data describing the record and 0 or more named Data Segments (usually just called “segments” – don’t ask why) usually containing binary arrays of data values. All datarecords in a given dataseries have the same set of keyword and segment names and associated record specific values. The dataseries description and the datarecords are maintained in a relational database called DRMS (Data Record Management System). DRMS is implemented as a set of PostgreSQL (see e.g. http://www.postgresql.org/) tables.

While the DRMS record contains the description of each datasegment, the information contained a datasegment is not stored in the database but is stored in “Storage Units” which are "owned" by SUMS (Storage Unit Management System). Storageunits are simply directories on disk or “tar”-files on tape. SUMS itself maintains tables in PostgreSQL to track storageunit locations on disk and/or tape. A storage unit will contain data for all datasegments for 1 or more records from the same series.

In summary:

A **Dataseries** consists of a set of:
Records which consist of a set of:
Keywords and
Segments each of which consists of:
structure information and a **storage unit** identifier

Normally one or more keywords are designated “prime keys”. The prime keys must together uniquely identify a record and are used to define the main index for the series. Any records with same set of prime key values are assumed to be different versions of the same record and normally only the most recent version of any record is easily available. Thus the current version of any record in a given series may be found by specifying the values of the prime keys for that series. All series have one pre-defined keyword called "recnum" which is has a

unique value for each record and is used for the main index in the case that no prime keys are defined. The implementation and implications of the version system is described near the end of this document.

In order to access a set of records from a series a descriptive “name” must be provided to select the desired records. We call that description a "Dataset Name". Thus, in JSOC/DRMS a dataset name is actually a database query. The DRMS dataset name rules have been defined to provide user friendly (well it is the goal) names that are easy to remember and use.

This document is concerned with the identification of data sets. Not so much with the logical or physical layout of the data or its storage or use but simply with naming conventions and the use of those naming conventions to identify specific data for purposes of processing.

Since in the JSOC/DRMS system a dataset is a set of data records selected by a query, we call the results of that query a Record Set (or just recordset). A recordset is the chunk of data that a program receives from and sends to the JSOC system. Most recordsets are expected to consist of records from the DRMS system however the programmers library and by extension the naming rules allow for access to data in a few other systems. To help differentiate the name rules for the different sources we call each source of datasets a “Catalog”. Thus the primary catalog of the JSOC system is the DRMS system.

In the DRMS catalog a dataset is, by definition, the result of a query. This is in contrast to the MDI DSDS catalog where a dataset is the contents of a directory and is usually a structured set of records from a single dataseries. In the MDI system the basic unit is a dataset while in the DRMS system it is a record. The unix/linux file system can also be viewed as a catalog and the JSOC recordset naming scheme allows for data records to be specified as simple files or directories of files in particular standard storage protocols (presently only simple FITS files are supported in this way.)

While one usually expects a dataset to be a set of records from a single dataseries this is not required. A dataset may consist of one or more sets of records even from mixed catalogs. This capability should be used sparingly. But in the naming discussion below remember that the final recordset is the concatenation of recordsets from multiple data specifications.

The naming rules provide several approaches to end up with a recordset. The “name” can be as simple (and normally is) a simple query resulting a compact list of one or more records from a single series or it can be a list of such specifications or it can be a file name where the file contains a list of such specifications.

In the case of a simple query, dataset name should be in such a form that it can be published in documents in a way so that the same data (possibly of later version) may be extracted at a later time.

The JSOC DRMS naming scheme is described here in a formal syntax. The following syntax is described using a BNF[1] format where entities are in angle brackets "<>" and literals are not enclosed except isolated non-alphanumeric characters which may be enclosed in "". Variants are separated by "|". Optional elements are in curly brackets "{}". Parentheses "()" may be used to group variants to make precedence specific. Recursion is explicit where allowed. The common use of square brackets "[]" for single options is not used here to avoid confusion with the use of "[]" as literals. Each entity is defined after it is first used. An ellipsis "..." is used to show either an incomplete list or a set of any text depending on context. "::=" is pronounced "is defined as a". "C"-like notation is used for non-printing characters.

A full DRMS name specification is at: http://jsoc.stanford.edu/jsocwiki/DrmsNames

Catalog

The term "catalog" is used here to denote an entire data collection. All of the MDI and solar group data managed by the MDI DSDS system is part of the DSDS catalog. Similarly all of the HMI and AIA data managed in the JSOC Science Data Processing at Stanford will be part of the DRMS catalog.

Series

The DSDS and DRMS catalogs each rely on the concept of a data series. In both cases a series is a set of data records where each record in the series has the same keyword attributes and the data stored in files has the same structure for each record in the series. In the DSDS case a blocking into DSDS datasets is also enforced. In both cases a record is the smallest unit that has named attributes. An example of a DRMS record would be a single raw filtergram or level-1 Dopplergram. DSDS records contain at most a single data array of a single variable or observable. DRMS records are allowed to contain multiple data segments each of which is an n-dimensional array of data.

DRMS Series names consist of two parts separated with a "." (period character aka "dot"). The leading part is a project name and is implemented as a Postgres namespace. This means that access privileges can be set for all dataseries in each project independently. The space of names in this first section will contain some reserved names, such as mdi, hmi, aia, wso, etc. For these reserved names the default ownership and permissions will be maintained in a table of reserved project names. Individual users each have their own project names consisting of two parts with the leading part indicating their group or institution and the latter part their specific identification (e.g. su_phil). The series part of the name (after the dot) should indicate the observable, the mapping or region size, the reduction level, etc., as appropriate. The cadence should also be specified as part of the name. E.g. "hmi.fd_V_50s" for HMI full disk Dopplergrams at a 50 second cadence.

[1] There are many "BNF" formats. See e.g. http://www.cs.man.ac.uk/~pjj/bnf/ebnf.html. Therefore the conventions used here must be explicitly defined.

DRMS series have a “prime index” which is defined in its series definition by a list of keyword names or defaulted to the absolute record number. The attributes named are used to construct a database index to allow rapid location of records identified by a value of its prime index. An example might well be using the attribute “T_OBS” as a prime index. In the case that the prime index keyword value is a real number (floating point type) it is often convenient to define a discrete mapping to a “slotted” axis to remove the uncertainty of exact matching of imprecise values. Each keyword used as a prime index component can have an identified type (e.g. time or latitude-bin) and depending on type some additional attributes to enable mapping of user-level convenient shorthand notation into efficient robust prime index values. For time types these would include a reference epoch and in the case of a slotted-time (i.e. TS_EQ) a step size or cadence. Some more details of the prime index associated keywords will be given later in this document.

DSDS datasets have the concept of version numbers (stored as level-numbers) such that there may be more than one version of a dataset identified by series and series-number. The same concept is implemented in the DRMS catalog but on the record level instead of on the dataset (or record block) level. So each record in a DRMS series may have multiple versions. Each record has an absolute internal record number as well as record ordering in the prime index space. Each “slot” in the prime index space may have multiple versions. A given record may be identified by either its prime index (in which case the most recently created version is referenced) or by its absolute record number.

Dataset

A dataset is a list of record sets (the fact that the dataset results in a single concatenated recordset as seen by the programmer should be kept in mind). A dataset name is a description of that collection. In the description here the term “dataset” is used interchangeably with “dataset name” where the meaning is obvious from the context. A dataset may contain records from one or more data series. A subset of the dataset that can be simply described as a group of records from a single data series is called a “record_set”. With this description, a dataset is a collection of record_sets separated by white space or a semicolon or a comma or a comment in the form of “#” through to the first instance of another “#” or newline character.

```
<dataset> ::=
     <record_set>{<dataset_sep>{<record_set>}}<ds_endmark>
<dataset_sep> ::= <ds_delim>{<dataset_sep>}
<ds_delim> ::= “;” | “,” | “#”...\n | “#”...“#” | \b | \t | \n
<ds_endmark> ::= “\0” | <end-of-file> (this may be simplified
to just white space)
```

Record Set

A dataset is a list of record sets. A record_set is a set of records from a single series from a single catalog. While a dataset will usually be drawn from a single series one may be a collection of record_sets from multiple series possibly from multiple catalogs. As used here a "catalog" is the top-level collection of data. The DRMS system implements the DRMS catalog. The MDI DSDS system is another catalog. JSOC exported FITS datasets will be considered as belonging to yet a third catalog. A record_set may be described by a name appropriate to its catalog or may be the name of a file containing a list of record sets. The expectation is that a complex set of records can be given in a file, perhaps one record_set per line, and that that filename can be used where a record set is needed. A dataset name containing white-space will usually need to be in quotes or other protection when used on a command line or in a script.

Record_sets of differing types coming from different "catalogs" as described below may be intermixed in a dataset. The first few characters of each record_set will be used to determine the associated catalog. The rule (so far) is: If a leading "**@**" is found the name after the "@" is a file pathname containing record sets else examine the leading substring terminated by the first instance of a "**:**", "/", "**[**", <ds_delim> or `<ds_endmark>`. If a "**:**" is found it is a non-drms structured dataset. If "**prog:**" it is DSDS dataset, if "**vot:**" then a VOTable based dataset *(TBR).* If a "/" is found it is a plain file or simple directory record_set. A simple directory containing an **overview.fits** file is treated as a DSDS dataset. Finally, and usually, if the substring is terminated by a "**[**" or **whitespace** or end of string it is a DRMS record_set.

```
<record_set> ::=
          <record_listfile_pathname> |
          <dsds_record_set> |
          <vot_record_set> |
          <pf_record_set> |
          <drms_record_set>

<recordset_listfile_pathname> ::= "@"<pathname>
```

Record sets may be provided in a text file using the same rules as on a command line. In particular they may be a compact query-like structure as described below or they may be a simple list of individual records. I.e. the "@filename" construct acts like an "include" file. Some levels of recursion are allowed but are limited in depth to prevent infinite looping. The meaning of characters preceding an "@" are TBD. Perhaps a URL should be allowed someday.

Example: A file named "/home/phil/magpair" containing the two lines:
prog:mdi,level:lev1.8,series:fd_M_96m_01d[5599]
hmi.M_lev1[2008.05.01/1d:96m]
used in a command line like:
Compare_mags in=@/home/phil/mapgair
would result in 15 magnetograms from MDI and 15 from HMI from 1 May 2008 in the dataset referenced through the command line keyword "in".

DSDS and plain file (pf_record_set) name rules are described elsewhere.

DRMS Record Set Definition

DRMS record sets are a subset of a single series and are specified by a seriesname and a record set specifier. If no record set specifier is present the record set consists of the entire series.

<drms_record_set> ::= <seriesname>{<record_set_specifier>}

<seriesname> ::= <namespace>.<series_specifier_name>

<namespace> ::= <name>

<series_specifier_name> ::= <name>

A seriesname consists of two parts, the namespace name and the series_specifier_name. The namespace takes the role of the DSDS "progname" but in the DRMS system goes further to provide a private set of dataseries that can be read by all but can only be changed by the owner of the namespace name. The owner may be an individual or a group.

A record set specifier identifies an ordered sequence of records. The requested specific sequence is described in a set of square brackets. The use of square brackets to delimit the record set specification is to be reminiscent of array indices in languages such as "c". The records may be specified by direct SQL-like queries or by "record_lists" which is are specifications of a sequence on the prime index of the series. In the case of a direct SQL query either specific records by absolute record number or an SQL "where" clause may be used. The query case will not be further described here (yet). A record_list is a set of records identified by values of that series prime index. A record list is then a set of prime_index_value_lists. Each such prime_index_value_list may be comma separated set of record ranges. There may be up to one prime_index_value_list for each keyword in the prime index.

There may be zero or more record_queries. The final record set is the logical **and** of the record_queries and record_lists present. Some examples will help – see below.

<record_set_specifier> ::=

{<record_set_specifier>}(<record_query>|<record_filter>)

<record_query> ::= ([?<sql_where_clause>?]|[:<recnum_range>])

<recnum_range> ::= <index_value>{"-"<index_value>}

<index_value> ::= #<integer_value>|#^|#$

<record_filter> ::=

{<record_filter>}[<prime_index_value_list>]

<prime_index_value_list> ::= {<prime_index_key>=}<index_list>

<prime_index_key> ::= <name>

<index_list> ::= {<index_list>“,”}<record_range>

Careful analysis of this rule shows that a record_filter or record_set_specifier is a chain of one or more “[…]” clauses. Each of these may be a range of absolute record numbers (recnums) or an SQL where clause or a list of values for one of the prime key components. For record_filters, any or all of the elements of the prime index may be used to identify the records. Each prime index keyword gets its own “[…]” clause. If the prime index contains more keys than are specified, the selection will include ALL matching records. Thus if the series is a set of lat-lon tiles as a function of time and only a range of times is specified then the selection will include all of the lat-lon tiles for each given time. Similarly if a subrange of one or more keys is specified then the selected set will include only that range for each of the prime keys. I.e. a specification acts as a filter limiting the range of records to be selected. An empty record set specifier selects the entire series. When multiple keys constitute the prime index the particular keys used in a record specifier must be clear either by direct identification or by context. If prime_index_key names are not given the index_lists will be matched to the components of the prime index in the order in which they appear in the series definition of the prime index. If a leading prime key element is not provided and subsequent elements are used without giving their names explicitly, the leading elements must have empty “[]” sections since the names are matched by the order in the prime key definition. *(NOTE: we are examining the option of allowing non-prime keys to be referenced using the <prime_index_value_list> format but have not yet adopted that option. We are also investigating a clean way to allow non-prime keys to also be indexed for database speed).*

Finally, we get down to a range of records specified by a range of values of a particular prime index component. The record_range may be given as a single record, as a “first” and “last” record in an interval, or as a “first” record and interval duration. In the case of an interval the minimum increment may also be specified.

<record_range> ::= <record_interval>{“@”<min_increment>}

<record_interval> ::= <record_id>{“-”<record_id>} |

<record_id>“/”<interval_duration>

A record interval is a “dash” separated first and last record id or in the case of a time axis it may be a set of records starting at a given record id and continuing “over” an interval of specified duration. The scope of the record_range is “closed-closed” when the first and last record_ids are given and “closed-open” when the first record and an interval are specified. (VERIFY) An example of full disk MDI magnetograms in the DRMS catalog for say 17 March 2005 would be:

mdi.fd_M_96m[2005.05.17/1d]

Note here that omitted time parts (hours, minutes, seconds) default to an obvious value. Then using the record_id axis_increment method described below we could have:

mdi.fd_V_lev18[3000d/1d]

which would mean the records for the day 3000 in the MDI epoch. In the DSDS catalog this would be specified as:

prog:mdi,level:lev1.8,series:fd_V_01h[72000-72023]

The epoch default is defined in the series definition (see below). In the MDI case the above spec expands to 24 datasets of 60 records each. In the DRMS case it expands to 1440 records. Of course the DRMS spec could just as well have been:

mdi.fd_V_lev18[72000h/24h]

The optional min_increment defines a minimum time step to allow undersampling of the target data series. Note this is unwise for oscillation studies which are observed with critical sampling. However a 27-day interval of 96-minute spaced magnetograms from HMI could be expressed as:

hmi.fd_M_lev1[2008.05.01/27d@96m]

or equivalently as

hmi.d_M_lev1[2008.05.01-2008.05.26_22h:24m@96m]

As can be seen in the examples here, there are several ways to specify a particular record. The above examples also hint at implied special knowledge about the type of the prime key values. This issue is discussed below. The basic DRMS implementation does not know about the keyword types except for the case that the type is “time” (one of the DRMS data types in addition to the usual int, float, etc.). When the type of the prime key keyword is “time” then the time may be entered in JSOC standard date format of e.g. yyyy.mm.dd_hh:mm:ss and that time will be converted into a double (the internal form of a DRMS time) representing seconds since the JSOC epoch of 1977.1.1_00:00_TAI.

Since the internal data types of keywords can be integer, floating, or string types the use of each of these types as prime keys must be addressed. Integer and string types are easy to deal with since they compare exactly with use entered values. Floating types however may not compare exactly depending on the computational history of a particular value that rounds to a particular printed value. Thus it is recommended that whenever a floating type is used as a prime key that all queries use a range to specify the particular record desired.

This is clumsy. For this reason we use the concept of a mapping onto an integer based “discrete”ized or “slotted” virtual axis for floating type prime keys.

The record selection may be made by specifying the prime index record number (i.e. slot), by value on the prime index equivalent axis is real space (e.g. a time) or by an increment along the prime index axis (e.g. a span of time). In all prime_index_value_lists the most recent version at the time of the record_id lookup is used. This will be described more below.

<record_id> ::=

<axis_index_value>|<axis_value>|<axis_increment>

Thus individual records may be specified in several ways: by index value which is an integer specifying the nth record along the prime index; by axis value which is an explicit value along the prime index axis; by specifying an increment since the series epoch along the prime index axis. These record_id variants will be described in order below.

Index_value

The index_value is the position on the slotted prime axis. It may be a basic step (cadence) in time for a time series, a bin number for latitude or longitude tiles, or a filter number, or NOAA AR number for examples. Good usage would dictate that a cadence in the prime index that is different from the "standard" step for a particular instrument/observable will be indicated in the series name. For an integer type the index_value for a record is by default the same as the axis_value except that the index_value is the same number prefixed by "#". So if an explicit axis type is not specified (see below) the same record may be addressed using the index_value notation or the axis_value notation with the same numerical value given.

The special values "^" and "$" refer to the smallest and largest existing index values for the current axis. Thus:

<index_value> ::= #<integer_value>|#$|#^

Axis_value

An explicit absolute value on a prime index axis may be specified. Some rules apply depending on the variable type of the prime keyword. These are as follows:

1. Integer types. No particular restrictions. The min and max values are restricted to the min and max allowed values for the associated data type. The integer types and limits are:
    a. char -128 127
    b. short -32768 32767
    c. int -2^31 2^31-1
    d. longlong -2^63 2^63-1
2. Floating point types. These are IEEE standard float and double types as 32-bit and 64-bit values. A printing conversion format is provided for all keywords and in the floating case this usually provides a limited precision to show the user. Since the database lookup uses exact bit matches and floating point values printed in limited precision involve rounding, the actual bit value may differ depending on the computation that is used to get a value. Thus whenever floating type keywords are used as prime keys we recommend specifying a range to identify the particular record desired. We advise against using floating types as prime keys in non-slotted keywords.
3. String types. These are ASCII strings (need to get correct spec here) which may contain blanks and other punctuation. If they are whole words, quotation is not needed. They are case insensitive! Thus the query value "Today" matches the record

with the value “today” and “Today” and “TODAY”. Care should be used since if all three of these examples are stored as records only the last one stored will be retrieved with a query formed as e.g. [ToDay].

4. Time. Times are a valid DRMS keyword type. Times are stored internally as doubles and thus are subject to the same warnings as other floating point types. An extra service is provided in the case of times in that DRMS defines both “internal” and “external” representations of times. The external form is the standard JSOC time format (see http://jsoc.stanford.edu/jsocwiki/JsocTimes) such as 2009.01.20_17:00_UT. The internal form is seconds since the JSOC epoch of 1977.01.01_00:00:00.0_TAI which is 1976.12.31_23:59:45.000_UT. A query in the form [2007.08.30/1d] will return the records for 30 August 2007. Times are by default given in UTC.

Axis_values are converted to index_values by the DRMS name parser guided by rules established when the series is defined.

The logical type of the virtual index axis must be known in order to interpret numerical values and match them with values on the prime index axis. These extended addressing types are available for string, integer, floating, or time types as desired. There are two basic methods, either specifying the value or distance from a reference value or “epoch” in the case of times.

Axis Increment

Slotted axis values can also be described as an increment from the base in a form that maps to the base units of the axis. Thus for a time axis an increment in time (e.g. 3000d) would Be interpreted as an offset from the epoch and for a Carrington axis an increment in degrees would be the same as the WSO/MDI Carrtime.

Sloted Axis Definitions and formats

For all slotted type prime keys several ancillary keywords must be defined along with the prime keyname itself. These keywords (all but one) must be explicitly described in the series definition (see jsd discussion in the JSOC wiki at http://jsoc.stanford.edu/jsocwiki/Jsd). There are several “types” of slotted axis supported. The list of types may be extended but at present includes equally spaced time series (TS_EQ), general slotted values (SLOT), enumerated lists (ENUM), and Carrington rotation and longitudes (CARR). In all cases the ancillary keyword names are formed by adding a suffix to the base prime keyword name. A keyword with the suffix “_type” must be present to define a slotted prime key axis. The _type keyword is itself a string which must contain one of the recognized types. If a recognized type is found an additional keyword with suffix “_index” is automatically created. The _index keyword is of type integer and is the actual keyword used as an index in the DRMS database. Depending on the type additional ancillary keywords must be present. These are listed below.

For a base prime keyword name of XXXX the required keywords are:

XXXX_type: string containing recognized type
XXXX_index: integer representing slot number

Type = TS_EQ requires
XXXX_epoch: Base epoch for axis, as a time or name of standard epoch
XXXX_step: Time increment defining the width of each slot
(XXXX_unit == seconds is implied.)
Axis values expressed as DRMS time (e.g. 2009.01.20_12:00_UT)
Axis increments are a number followed by a modifier, one of s,m,h,d e.g. 2000d was 24 June 1998 using the MDI epoch.

Type = SLOT requires
XXXX_base: Base reference value for axis as a double precision number.
XXXX_step: increment defining the width of each slot as a double precision number.
XXXX_unit: string describing units along the axis. Used for increments.
Axis value is simply a number.
Axis increments are a number with a suffix string of XXXX_unit.

Type = ENUM requires
XXXX_vals: A list of string values defining the names of each slot. These are case insensitive. If numbers are used as list values they should be enclosed in quotes.
Axis values are strings from the set defined in the XXXX_vals.

Type = CARR requires
XXXX_step: Increment in degrees. Base is always WSO/MDI Carrtime=0.
(XXXX_unit == degree is implied)
Axis values are in the CT:2000:20 format, CT:<rotation>:<degrees>
Axis increment values are in the form CT:123456. Axis increments can not be used for the first few rotations since the value of the rotation number is used to differentiate rotation numbers from increments in degrees.

PERHAPS:
Type = TS requires
XXXX_epoch
XXXX_step
Axis values are times. Increment time intervals. But expected to be sparse
so axis indexing with prime key record number (#ddd notation) not expected
To be very useful. Perhaps make step be 1-second then toss AIA images into slots to avoid missing them. I.e. make step the same as the ascii resolution used to specify images. Perhaps 0.1 second. Think of it as rounded times.

Type = SPH requires no additional keywords but allows a prefix of “l” (ell) to represent spherical harmonic degree. Further definition of this option needs to be made to allow

spherical harmonic tiles to be addressed in a reasonable manner. Perhaps just as a list of strings such as “30-39” to contain l=30 to l=39.

For all slotted axis calculations the ancillary keyword values are used to map between the axis value and slot number. The rounding is closed-open such that half way between slots rounds up to the larger slot number.

The ancillary keywords including _type are all CONSTANT scope keywords.

Standard Epochs for slotted time axes

The XXXX_epoch values can be expressed directly as a time of as a string that matches one of the standard epochs defined as follows:

<epoch_name> ::=

JSOC_EPOCH |

MDI_EPOCH |

WSO_EPOCH |

TAI_EPOCH |

MJD_EPOCH

Where:

JSOC_EPOCH == 1977.01.01_00h:00m:00s_TAI
MDI_EPOCH == 1993.01.01_00h:00m:00s_TAI
WSO_EPOCH == 1601.01.01_00h:00m:00s_UT
TAI_EPOCH == 1958.01.01_00h:00m:00s_TAI
MJD_EPOCH == 1858.11.17_00h:00m:00s_UT

Versions

In the JSOC DRMS for series with prime keys defined (almost all series) the always present “recnum” absolute record number is used to track the most recent version of each record. All records with the same prime key values are treated as different versions of the same record. Each time a record is added to a series, recnum is incremented. So for each set of records with the same prime key values the one with the highest recnum value is the most recent one added and is then the current version.

Since records can be selected in several ways the interaction of the version logic with the selection specification must be understood if the desired results are to be obtained. The basic rule is that the version check is made only when at least one prime key filter is provided. Since each selection clause adds to the record selection filtering in the sense of a logical AND the filtering can be done in any order. Since it is hugely more efficient in the database

to first select based on indexed quantities and the prime keys are indexed, the selection based on prime keys is done first. After the prime key selection is done, the highest version is selected. Only then are any general WHERE clauses executed (the [? … ?] notation).
If no prime keys are used for filtering then the version checking is not done. This can lead to some unexpected results. Suppose a series has only 2 keywords, A and B with A as the only prime key. Now suppose there are only a few records containing:

Recnum	A	B
1	50	red
2	51	blue
3	51	pink
4	52	white
5	53	blue

The query [50-53] will yield records 1,3,4,5.
The query [? B=blue ?] returns records 2 and 5 but
The query [^-$][? B=blue ?] returns only record 5

Queries using [^-$] should NOT be used on large series. In general the more the request can be limited by specifying a range of prime keys the better. The performance difference can be dramatic as in 30 minutes vs 30 ms in the case of our expected largest series (2 second cadence for 5 years).

NOTE this is different from the present implementation which goes to pains to do the version test even when no prime keys are specified in the query.