USDA United States Department of Agriculture

Natural Resources Conservation Service

Soil and Plant Science Division

National Soil Survey Center

NASIS CVIR Language Manual

Scripting Language for NASIS

Calculations, **V**alidations, **I**nterpretations, and **R**eports

NASIS 7.4 and

Web Soil Survey Rule and Report Manager

U.S. Department of Agriculture
Natural Resources Conservation Service

Original text by Gary Spivak: February 1, 2011
Updated by Kevin Godsey: March 1, 2018
Issued June, 2019

Nondiscrimination Statement

In accordance with Federal civil rights law and U.S. Department of Agriculture (USDA) civil rights regulations and policies, the USDA, its Agencies, offices, and employees, and institutions participating in or administering USDA programs are prohibited from discriminating based on race, color, national origin, religion, sex, gender identity (including gender expression), sexual orientation, disability, age, marital status, family/parental status, income derived from a public assistance program, political beliefs, or reprisal or retaliation for prior civil rights activity, in any program or activity conducted or funded by USDA (not all bases apply to all programs). Remedies and complaint filing deadlines vary by program or incident.

Persons with disabilities who require alternative means of communication for program information (e.g., Braille, large print, audiotape, American Sign Language, etc.) should contact the responsible Agency or USDA's TARGET Center at (202) 720-2600 (voice and TTY) or contact USDA through the Federal Relay Service at (800) 877-8339. Additionally, program information may be made available in languages other than English.

To file a program discrimination complaint, complete the USDA Program Discrimination Complaint Form, AD-3027, found online at How to File a Program Discrimination Complaint and at any USDA office or write a letter addressed to USDA and provide in the letter all of the information requested in the form. To request a copy of the complaint form, call (866) 632-9992. Submit your completed form or letter to USDA by: (1) mail: U.S. Department of Agriculture, Office of the Assistant Secretary for Civil Rights, 1400 Independence Avenue, SW, Washington, D.C. 20250-9410; (2) fax: (202) 690-7442; or (3) email: program.intake@usda.gov.

USDA is an equal opportunity provider, employer, and lender.

Contents

Scripts

SQL Syntax

Report Syntax

Additional Information

Appendices

Scripts

Introduction

This guide provides the NASIS user an understanding of SQL as it relates to various uses in NASIS. The Web Soil Survey Report Manager is also based on the NASIS Sequel-Informix format. However, the DocBook syntax must be followed for reports to be properly formatted. SQL is used to write queries, reports, properties, calculations, and validations. An understanding of the NASIS data structure (tables and columns) is required before using SQL.

This document is to be used with the NASIS website References. The following reference documents are necessary for understanding the Query and Report writing process.

The "Tables and Columns" document, which identifies the NASIS tables and its columns; the "NASIS CVIR Language Manual, Scripting language for NASIS Calculations, Validations, Interpretations and Reports;" and NASIS Data Structure Diagrams and Data types and Comparison Operators chart, which is in this document.

Overview of CVIR Scripts

In NASIS, all Queries, Reports, Calculations, Validations, and Properties contain a script, which is a set of instructions for reading data from the database and using the data to produce some result. All these scripts have a common structure but have various options that are specific to their function. This reference manual contains the complete specifications for CVIR scripts. NASIS 7.0 and later versions are a combination of Sequel and Informix script writing. Most of the SQL follows standard SQL syntax in the SELECT, FROM, and WHERE clauses. Informix syntax is used in the SORT and AGGREGATION section of the SQL.

The major parts of a CVIR script are Query, Data Manipulation, and Output.

Query

Instructions to read data from the database, written in a variant of SQL (Structured Query Language, an international standard for working with relational databases, often pronounced "sequel"). A simple CVIR query looks like:

```
EXEC SQL SELECT musym, muname from mapunit, lmapunit
WHERE JOIN mapunit TO lmapunit;.
```

The "Query" section uses the SQL KEYWORDS (Sequel syntax)

SELECT columns
FROM tables
WHERE conditions are met

The query can also contain optional sections, such as “sort” and “aggregation” (Informix syntax).

Sort By muiid
Aggregate Rows by Muiid Column nuacres SUM.

Data Manipulation (Informix syntax)

A series of instructions for working with the data to produce new data values, using mathematical formulas, if-then-else logic, and other operations. The “Data Manipulation” section includes several tools used to transform the data “output”. These include the following statements:

CODELABEL/CODENAME
DEFINE
ASSIGN
DERIVE
INTERPRET
PARAMETER

Examples of data manipulation statements

```
DEFINE complabel IF ISNULL(localphase) THEN compname
     ELSE compname || ", " || localphase.
DEFINE dt TODAY.
DEFINE legend_name  areaname || ": " || legenddesc.
DEFINE mu_namemuname.
```

The Data Manipulation phase of the report allows for data pulled from the SQL to be transformed to another value or class. The Data Manipulation section requires the use of the OUTPUT section in a report. If no output sections are in the report, then the default output is generated. It only displays columns from the first query in the order of selection. The headers can be altered with an alias in the select clause.

Output

Instructions for producing some result. In a report script, this includes the specifications for laying out the report; for a calculation script, it specifies which columns of the database record will be updated; and so forth. Text formatted reports follow Informix syntax, but the HTML output follows DocBook format; most standard HTML tags are allowed. Typical output statements for a report are:

```
TEMPLATE mapunit
     AT LEFT FIELD WIDTH 20, FIELD WIDTH 8 SEPARATOR "|".
SECTION
     DATA
          USING mapunit muname, musym.
END SECTION.
```

Examples

```
PITCH HORIZONTAL 15 VERTICAL 8.  PAGE LENGTH 80.

TEMPLATE basic SEPARATOR AT LEFT FIELD WIDTH 10, FIELD
WIDTH 90. TEMPLATE head SEPARATOR AT LEFT FIELD WIDTH 10,
FIELD WIDTH 25.

HEADER INITIAL
AT LEFT areaname WIDTH 75;
AT 80 "Print date: ", dt WIDTH 10. AT LEFT "Soil Map
Legend".
END HEADER.

HEADER
AT LEFT "Soil Map Legend".
END HEADER.

SECTION main HEADING
SKIP 1 LINE.
USING head
"Map \n symbol" ALIGN CENTER, "Soil name" ALIGN RIGHT.
USING basic.
DATA
USING basic
musym INDENT 1,
mu_name INDENT -1.
END SECTION.

SECTION WHEN LAST OF liid DATA
USING basic. NEW PAGE.
END SECTION.
```

CVIR scripts are stored in tables in the NASIS database and are assigned ownership in the same way as data in the soils tables. Anyone can run a CVIR script, but to edit a script you must be a member of the group that owns the script and you must check it out from the server. The scripts are in the Explorer area (left side) of the NASIS screen. They are organized by the type of script and the NASIS site. Menu options are provided to run a script or to open it for viewing and editing.

The CVIR Syntax Reference section of this document describes each CVIR statements in detail. The section is arranged in alphabetic order and notes what types of scripts each statement can be used in. This information helps you get the syntax right but doesn't explain the overall concepts. The next few sections of this document give guidelines on writing scripts. See the section below for more information on SQL syntax.

Data Flow in CVIR Scripts

A script specifies a set of actions to be performed but not necessarily the order in which they will be performed. The CVIR processor collects all the statements of a kind and processes them as a group at the point in the data flow where they are needed. The sequence is:

1. Configuration statements are processed just once and before anything else happens. These statements don't produce any actual data but instead specify conditions for other statements to work with. They include:
   ACCEPT
   PARAMETER
   BASE TABLE
   PAGE, MARGIN, FONT, and PITCH
   TEMPLATE
2. Data input statements provide the data that the script works on. Most scripts include the EXEC SQL statement to get data from the database. The INPUT statement can be used to read data from a file. There are many rules about the interactions between data input statements when the script contains more than one input statement. These rules are described in the Syntax Reference section. Many times, a report consists of a main query (initial data) and then several subqueries (additional data). These subqueries can be linked to the main query or can be independent of it.

3. Data from the query can be stored in variable in the NASIS database or in Temporary Tables.

4. References to other scripts are processed next. The DERIVE statement obtains data from a Property script, and the INTERPRET statement generates interpretations that can be displayed in a report.

5. The next step is to get data from derived statements. These are statements that can have parameters that are passed to subreports. Again, these variables can be stored in the database or in variables that can be used later.

6. Data manipulation is the next step. This is done with the DEFINE and ASSIGN statements, which can include mathematical formulas, if-then-else conditions, and other functions. The result is a set of variables whose values can be used in the next step.

7. The final step is output of data, and this varies depending on the type of script. In a Report script, the output specifications define how the information will be formatted. In a Calculation script, the output specifications identify what fields in the database will be updated. A Validation script specifies the messages to be produced when error conditions are found. A Property script has no output section at all.

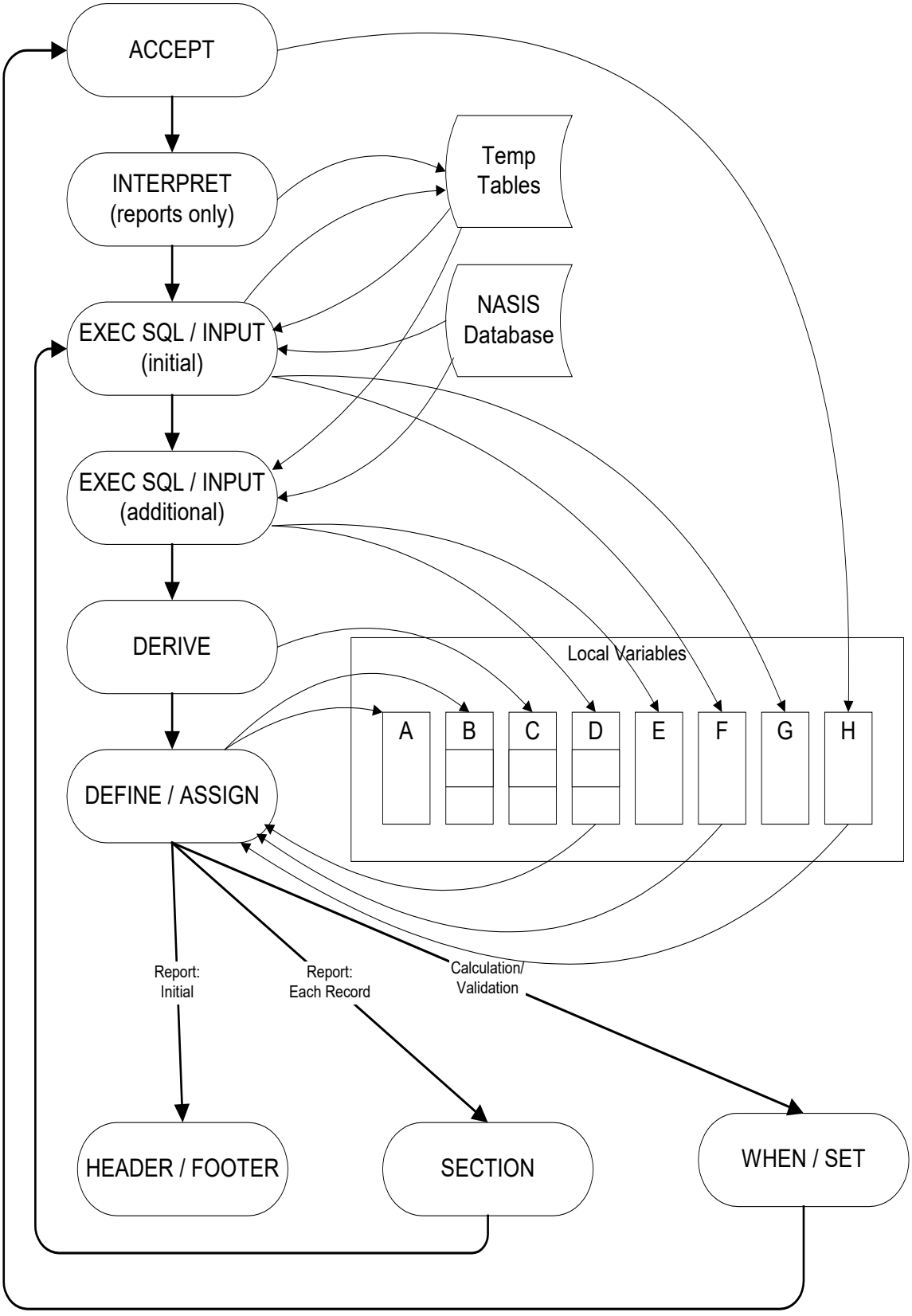
8. In the Output section, other reports can be added using Include statements (discussed later).

9. After completion of the first 8 steps, the process returns to step 2 and repeats until there is no more data to process.

10. Header and Footer sections are processed only once, while data sections are repeated for each cycle of the report. If the component table is the base table, the report will repeat for each component in the selected set until all components are processed.

Following is a diagram showing the flow of data from the Accept statement through to the output.

Data Flow Diagram for CVIR Scripts



Query Scripts

In NASIS, a Query (with a capital Q) is a script used to find data for use in a NASIS session. A Query can be run against the national database to identify data to be downloaded to the local (workstation) database, or it can be run against the local database to identify the data that will appear in the table editors and reports (known as the *selected set*).

A Query script is the most minimal of all CVIR scripts. It consists of exactly one, partial SQL query. It does not even use the EXEC SQL prefix as in other CVIR scripts. The only parts of a SQL query it uses are the FROM and WHERE clauses. When a Query is run, NASIS produces the full SQL query using the script plus any run-time options selected by the user, such as target tables and parameters.

Details on the CVIR variant of SQL are in the Syntax Reference section under EXEC SQL. The key features of a Query script are:

- All the tables listed in the FROM clause are candidates to be target tables when the Query is run. For each target table the user picks, a SQL query is constructed to select rows in that table. Additional rows linked to the target table are then found to fill out the selected set or download list.
- Query parameters can be specified in the WHERE clause by using a comparison with a question mark, such as "areasymbol =?". When the query is run, NASIS creates a field for the user to enter an area symbol. The query automatically looks up "areasymbol" in the metadata to determine the data type of the areasymbol and whether the areasymbol has a fixed choice list (domain).
- When a query is sent to the National database, many more tables and data are downloaded to make sure that all linked data is included. However, sometimes a query needs to be run several times with different tables targeted so that all relevant data is in the local database.

To become adept at writing queries, the user must have knowledge of the Structured Query Language database language. SQL, as it is commonly referred to, was created by IBM in the early 1970s as a unified language for defining, querying, modifying, and controlling the data in a relational database.

Over 75 different flavors of SQL are now in commercial use. NASIS originally used the Informix database but now uses the Microsoft SQL Server database. Although the basic SQL structure is standardized between commercial databases, there are dialect differences. This document focuses on the SQL Server dialect and how it is used with the various soils databases. SQL is used in NASIS, the Soil Data Mart, the Soil Data Access site, and Web Soil Survey. Understanding SQL allows the user to query data or write reports from these databases and sites. DocBook formatting is the best way to ensure that your report looks like a Web Soil Survey report.

Property Scripts

After the Query, the next simplest type of script is the Property script, which has several uses in NASIS. As the name suggests, a Property script is primarily used to get soil property data from the database. It typically contains all the logic needed to select data from the appropriate tables, aggregate the data to the required level, and perform other needed transformations. Putting all three parts in a Property script provides consistency in the use of a particular soil property.

Property Scripts are Used:

- In the Evaluation portion of an Interpretation. In the Evaluation Editor, the user picks exactly one Property to be the source of data for the evaluation function. When used for this purpose, the Property must define one or more of the variables "low", "rv", and "high" for use by the Evaluation. The type of data and number of these variables must agree with the Data Type and Modality fields entered on the General page of the Property editor. The Property cannot use the ACCEPT statement to take parameters.
- In the DERIVE statement of another CVIR script (Report, Calculation, Validation, or another Property). When used another script, the statement is not limited to the variables "low", "rv", and "high" because the DERIVE statement specifies which Property variables it is expecting. The DERIVE statement can also pass parameters to the Property's ACCEPT statement.
- In the Interpretations Editor with the Run button. This use is just for testing a Property. The user first highlights a row in the Property's base table and then clicks Run. A page of output is displayed. The page lists all the variables defined in the Property script and the values they take on when the Property is run with the selected row of data.

Key Features of a Property Script:

- Properties used in Interpretations have a BASE TABLE component.
  - *Important: There must be a BASE TABLE statement to specify for which database table the queries in the script are run.*
- The script can include one or more queries. The queries are run in the context from which they are called. For example, when called from an Evaluation the queries return data for the Component record being evaluated; and when called from a CVIR script, they return data for the row of the Base Table that the caller is operating on. The CVIR engine adds an extra condition to the query's WHERE clause to limit the results to the current row of the base table.
- The script can invoke other Property scripts using the DERIVE statement. For further information, see the Syntax Reference section.
- The script can include DEFINE and ASSIGN statements to create new variables or modify the values of variables supplied by the queries.

Calculation and Validation Scripts

Calculations and Validations are scripts that automate complex data-entry or data-checking procedures. A Calculation is an approved, standard procedure for deriving data values using data previously entered in the database. A Validation is an approved, standard procedure for checking that various data in the database are consistent with each other, fully populated, or both.

To Use a Calculation or Validation Script:

- Open the Table Editor to the Base Table defined in the script.
- Select one or more rows in the table.
- Locate the Calculation or Validation in either the Explorer or Editor Window and click Run.
- If running a Calculation, check the options as needed to allow the calculated results to replace data that was manually entered (tagged M) or that was entered prior to the time source tracking was established (tagged P). If these are not checked, a Calculation can only replace empty fields or fields that have previously calculated values.

Key Features of Calculation and Validations Scripts:

- The script can contain all the parts described above for Property scripts.
- A Validation script has one or more WHEN statements that produce messages when a specified condition occurs. The messages are displayed in the NASIS message window along with a link to the row that produced the message.
- A Calculation script has one or more SET statements that identify the fields where calculated results will be placed. A Calculation script can also use WHEN statements to produce messages about problems with the calculation.
- Calculations and Validations may be created or edited only by people belonging to an administrative (NSSC Pangaea) group in NASIS. If the script is marked "Not Ready for Use," only a member of the group that owns the script can run it.

Report Scripts

A Report is the most complex type of CVIR script because there are many details that must be specified to produce the exact desired output format. There are two independent styles of report generation: text based and XML based. They are discussed separately. Also, there is an option to run a report on either the local or the national database, which can produce different results.

To Use Reports:

- Open a Report from the Report Explorer to see the script and other details in a Report Editor window.

- Run an open Report from the Report Editor. To run a Report without opening it, select it in the Report Explorer and pick a Run option from the Explorer menu. Select either Run Against Local Database or Run Against National Database. To avoid producing too much output, see the notes below about running against the national database.
- The output of a Report is placed in a temporary file on your computer and then opened with the standard application for the type of file. For example, the standard application for HTML output is a web browser (such as Internet Explorer) and the standard application for text (TXT) output is commonly Notepad. In Windows 10, you can configure the application for each file type using Control Panel > Folder Options > File Types.

Key Features of a Report Script:

- The script can contain all the parts described above for Property scripts, plus output specifications.
- A script that contains only a query (EXEC SQL statement) produces a standard default report output consisting of a simple table. The column headers are in order of the select-clause list. This default formatting is convenient for a quick review of the data listing. The headers can be altered with an alias. The aliases are displayed in the header only if the field is altered. The easiest way to alter a field—and thereby display the header—is to add zero to a number (`comppct_r + 0 AS component_Percentage`) or concatenate a blank quote to a text field (`muname +"" AS Map_unit_Name`).
- The string concatenation operator is the plus sign (+) in SQL. The double pipe (||) is the concatenation for DEFINE statements. You can combine, or concatenate, two or more character strings into a single character string.
- If the plus sign is used with two numeric values, then the values will be added together.
- You must cast numeric values into a character string to concatenate numbers to text strings.

Note: At this time, you cannot concatenate the map unit symbol and the map unit name because they are set to two different data types.

The INTERPRET statement can be used to generate interpretations for inclusion in the report output.

- For text-style reports, the default layout is an 8.5- by 11-inch page with ½ inch margins and 12 point font. Page layout statements can be used to change these specifications. For XML- or HTML-style reports, page specifications are not used because the browser and style sheet control the final appearance of the report.
- The TEMPLATE statement can be used to define the layout of a line of report output. This makes it easy to apply the same layout to many output lines, which produces a consistent appearance.

- The SECTION statement provides a way to group the report output statements and specify where they are used. A Section represents a block of report output that can appear at the beginning or end of the report, at a point where some data changes, or repeatedly.

Reports can be loaded into the Report "table," allowing the user to query for and manage reports in a table format.

Reports can be "Run Against the Local Database," "Run Against the National Database," or "Run Offline Against the National Database," Using the "offline" option allows lengthy reports to be run on the server instead of your local machine. The advantage of the offline option is that it releases the NASIS screen, allowing you to continue working in NASIS while the report is running on the server. When the report is completed, the server sends you an email with a link to the completed report.

The metadata reports and CVIR guide are necessary references for report writing. Find these materials on the NASIS website and keep them available. Links to these files are in the Documentation folder in NASIS in a report named "NASIS Help."

NASIS Reports contain 3 major parts:

- QUERY
- DATA MANIPULATION
- OUTPUT

Text-Style Reports

Text-style output is the type of report produced by NASIS 5.x and earlier. It consists of lines of text in which each line is just a series of characters and each character takes up a fixed amount of space. When a series of lines with consistent layouts are produced, the characters line up to produce the appearance of columns in a table. Often a vertical bar symbol (|) is used to separate report columns. Text-style output generally does not look right when printed with a proportional font.

Text output does not have to be in tabular format. It can look like paragraphs or lists, because each line of output can have its own format. The output can also be stored in a file and imported into another program, such as a spreadsheet, using either a fixed column width or variable width and delimiters. Existing reports in NASIS can be used as examples for any of these output formats. Copying and modifying an existing report is much easier than starting from scratch.

The AT command is the key to producing text-style output. It specifies where on the line each piece of data will be placed and how it will be formatted. The Reference section below describes numerous options.

XML-Style Reports

XML-style output was developed for Web Soil Survey and is included in NASIS 7. XML (eXtended Markup Language) is a standard data format. It is called a "markup language" because it contains markings, or tags, mixed with the text. A tag is a name, and possibly some other attributes, enclosed in angle brackets; for example, `<table>`. A tag typically describes how the text that follows the tag is used, or what type of text it is. XML output can be used in a variety of ways, including as an import to other programs or as a source for transforming into formatted reports.

XML output is produced with the ELEMENT command. A report cannot use both ELEMENT and AT commands. An ELEMENT always has a name and may also have attributes and content.

- An element name must correspond with a standard XML tag for the type of output you want to produce. For example, a formatted table begins with a `<table>` tag, which is specified in a NASIS report as `ELEMENT "table"`. The standard tags are listed in Appendix 5: HTML Formatting.
- Many XML tags have standard attributes that modify the output appearance. For example, a table can have borders drawn between cells with the tag `<table border="1">`. In a NASIS report, this is written as `ELEMENT "table" ATTRIB("border", "1")`.
- Almost all XML tags have content, which could be other tags or just data to be output. The content is preceded by a tag (such as those just described) and followed with a closing tag. Closing tags include the tag name prefixed with a slash. For example, a paragraph in XML might be `"<para>This is some information to print.</para>"`. The corresponding NASIS statement would be `ELEMENT "para" "This is some information to print."`.
- A plain ELEMENT command produces both the opening and closing tags. Sometimes, it is not possible to include all the content you want in a single ELEMENT command. In such cases, you can use `ELEMENT OPEN "name"` to produce just the opening tag. Then, later in the report script, you must use `ELEMENT CLOSE "name"` to produce the closing tag.

HTML-Style Reports

NASIS allows the report output format to be specified as HTML, which is a variation of XML. To produce reports that look like Web Soil Survey reports, use the elements and attributes listed in the DocBook XML section of Appendix I. These elements and attributes are converted automatically to HTML, which is the standard for displaying in a web browser. If you are familiar with HTML, you can use regular HTML tags as NASIS elements instead of DocBook

XML. Note, however, that HTML tags do not work in the Web Soil Survey Rule and Report Manager.

HTML output is like XML but does not show the markup tags. These two types of reports are created the same way and are discussed only in this section. HTML (HyperText Markup Language) is another example of a standard "markup" formatting language. It too contains markings, or tags, mixed with the text. HTML reports can easily be copied and pasted into other programs, such as Word, Excel, and Access.

Important: HTML tags do not work in the Web Soil Survey Rule and Report Manager.

HTML output is produced with the ELEMENT command. A report cannot use both ELEMENT and AT commands. An ELEMENT always has a name and may also have attributes and content.

A plain ELEMENT command produces both the opening and closing tags. Sometimes, it is not possible to include all the content you want into a single ELEMENT command. In such cases, you can use `ELEMENT OPEN "name"` to produce just the opening tag. Then, later in the report script, you must use `ELEMENT CLOSE "name"` to produce the closing tag. The end closing tags must be placed in the reverse order of creation; otherwise, you will receive an error message that the tags do not match.
ELEMENT "p" is a shorter version of "para".
ELEMENT "pre" will retain blank spaces and formatting in the output.

Examples

```
ELEMENT "para" "This is some information to print.".
```

Example error message:

> ERROR
> While running Report Script "test error messages"
> The "body" start tag on line 2 position 2 does not match the end tag of "html".
> Line 454, position 3.

More errors are discussed in Appendix IV: Common Error Messages.

Running Reports Against Local or National Database

When a report is run against the local database, the report normally uses only data in the *selected set*. This is convenient because a report can be designed without parameters and then run with different selected sets to generate different output. But, it also means that Queries need to be run to get data into the selected set for every table used in the report. EDIT specifies which data set the query is run against. The default setting is EDIT and does not have to be in the script.

A report can also be written to use any data in the local database, regardless of whether they are in the selected set. To run the report against data that is not in the selected set, put the word REAL before the table name in the FROM clause of the report query. A query can have a mix of REAL tables and standard tables.

The term REAL in front of the table name links the table in the local database instead of the selected set. When the query is run against the national database, the term REAL is ignored. "REAL" needs to be on all the tables that are **by default** in your local database (e.g. area, areatype, geomorphic tables, etc.).

When a report is run on the national database, there is no selected set; therefore, all tables act as if they had the term REAL in front of them. Because the national database is very large, careful planning is important to avoid excessive run time and output. A report designed to work on the selected set is typically not appropriate for the national database. The PARAMETER statement should be used so that the user can specify criteria to limit the report output. The term "NAT" is added to the end of the query or report to identify it as a national report script.

Reports can be run on the national database either normally or offline. When a report is run normally, the program waits for the report to complete and then displays the output. When a report is run offline, you send the request to the national database and continue working in NASIS. When the report completes, you receive an email with a link to the report output. Running national reports in the normal way is subject to limitations on time and number of rows returned. These limitations often cause reports to not finish. Offline reports can run longer.

SQL Syntax

SQL Syntax Elements

A SQL statement contains several elements. SQL has "Keywords" that have special meaning. Although SQL is not case sensitive, the keywords are typically entered in UPPERCASE. This capitalization is done for organizational purposes only. SQL statements also contain identifiers that are the names of the databases, tables, and columns. Typically, identifiers are entirely in lowercase. Statements also contain operators or functions used for comparisons or mathematical equations. The operator can be used for arithmetic (+ or -) or comparisons (> or =), or it can be used for logical (AND, OR, NOT) or aggregate (MAX, MIN, SUM, COUNT, AVG) functions.

Keywords

The basic SQL statement consists of 3 key words:

- SELECT (column)
- FROM (table)
- WHERE (condition)

The SELECT clause:

- Specifies the columns (e.g., musym, muname, mukind) to be retrieved.
- Requires each column to have a unique name.
- Allows for expressions that must follow normal SQL syntax (e.g., `sandtotal_r + silttotal_r + claytotal_r AS particle_size`).
- Requires an alias (e.g., "particle_size") to be used to provide a unique name if expressions are used in the SELECT statement.

The FROM clause:

- Specifies all the tables used in the query.
- May specify aliases and joins.
- Indexes aliased table names faster.
- May contain predicates (conditions on the data).

The WHERE clause:

- Filters which rows to use in the FROM clause.
- Uses normal SQL conditions.
- Uses the NASIS "JOIN *table* TO *table*" syntax to simplify writing join conditions.
- Requires two tables in a JOIN condition to have a relationship.

- Has a best practice in which you join the tables and have conditions in the FROM clause.

Example

```
SELECT nationalmusym, muname
FROM mapunit
WHERE muname = "Harney silt loam, 0 to 1 percent slopes";.
```

Identifiers

The NASIS and SSURGO metadata reports, found on the appropriate websites, describe the identifiers needed for SQL statements. The document "Tables and Columns.pdf" provides the list of tables and the columns within each. These documents are designed to provide the user with information necessary to write SQL statements.

The report named "Scripting Parts" in the Documentation folder has many of the most commonly used scripts in the proper syntax.

Conventions Used in this Guide

CVIR script statements are arranged alphabetically in this technical guide so you can find them more easily. The descriptions include sections for Syntax, Used In, and Examples.

Syntax

The syntax is described in a formal notation, using the following conventions:

- Braces { } enclose a set of alternatives, of which one must be chosen.
- Any portion of a definition in brackets [ ] is optional.
- When an ellipsis (...) follows the brackets, the optional part can be repeated.
- The symbol ⇒ means "is defined as" and defines a term that appears in a previous statement definition.
- Punctuation in bold print is a required part of the statement.
- Keywords in the CVIR language are shown in sans-serif capital letters (e.g., DEFINE), but the interpreter is not case sensitive for keywords or variable names.

Used In

The Used In line identifies the type of scripts in which the statement may be used. The types listed are Report, Subreport, Property, and Calculation. Validation scripts use the same statements as Calculations.

Examples

The examples show sample code.

ACCEPT

Syntax

ACCEPT variable [, variable] ... .
variable ⇒ *name*

Used In

Subreport, Property, Calculation

Examples

```
ACCEPT datamapunit_iid.
ACCEPT top_limit, bottom_limit.
```

The ACCEPT statement defines variables that are passed into the script. These variables can be used in expressions to calculate values for other variables. They can also be used in the FROM and WHERE clause of a query by writing $name, where *name* is the name of the variable. This creates a parameterized query and is discussed under EXEC SQL.

The first example of the ACCEPT statement could be used in a subreport. The value of a key column, such as datamapunit_iid, might be passed by a higher level report. The subreport would use the value in a query to find data related to the datamapunit being processed in the higher level report.

The second example might be used in a Property script. In this case, two variables are passed by some script that calls the property, and they could be used to perform some calculation in the property script. Any variable names may be used because they do not refer to database columns.

The number of parameters passed by the caller must equal the number of variables in the ACCEPT list. The type and dimension of these variables are not predefined, so they are determined by the values passed by the caller. For a Property, the primary key column of the base table acts as if it were in the ACCEPT list, even if the property script has no ACCEPT statement. The primary key column is used to ensure that the property is providing data for the same record as the script that calls it. Consequently, a calling script and it's called property scripts must use the same base table.

BASE TABLE

Syntax

BASE TABLE *table-name*.

Used In

Report, Property, Calculation

Example

```
BASE TABLE component.
```

When a CVIR script and its associated properties have several database queries, automatic coordination is performed between the queries by specifying a BASE TABLE. For example, if the base table is Component, the automatic coordination ensures that each query provides data for the same component during a cycle of the script. A report script requires a base table if the script includes an ACCEPT statement, more than one query, or DERIVE statements.

In a report script, the processing cycle is determined by the aggregation specifications in the first report query, but the base table provides the key used to synchronize queries and properties. For example, the first query may include a statement like:

AGGREGATE ROWS muiid, coiid

In this example, the component id (coiid) is the lowest level of aggregation, so the report performs a cycle for each component. Normally the base table for this report, if needed, would also be component. Deviation from this norm is an advanced report capability that requires careful testing to ensure that the report works correctly.

A Calculation or Validation script differs from a report in that it performs one complete execution for each base table row that has been selected by the user and has been checked out for editing. It accepts key input values from the current base table row, and it stores the calculated data elements in the same row.

Important: A Property script requires a BASE TABLE to coordinate its query with those in the calling script. The Property script does not need a BASE TABLE if it uses a parameterized query.

The *table-name* used in the BASE TABLE statement must be one of the tables defined in the NASIS metadata. Either the logical name or the physical name may be used, but the name must be spelled the same as in the queries.

DEFINE

Syntax

DEFINE variable [expression] [, expression]… [initialization] .
ASSIGN variable expression [, expression]… .
expression *(see next page)*
initialization *(see next page)*

Used In

Report, Property, Calculation

Example

```
DEFINE status CODENAME (mustatus).
ASSIGN status status || " mapunit".
```

The DEFINE statement defines a variable for use in a CVIR script. Each defined variable name must be unique within a script, and each must be different from the names of columns in the input queries.

The ASSIGN statement recalculates the value of a variable that was defined in a previous DEFINE, DERIVE, or EXEC SQL statement. No alias is used with ASSIGN, the transformation literally replaces the previous data for the specifically named field.

In the following example, *compkind* is listed in the SELECT command. The CODENAME for this field could be created in the SELECT command, as a DEFINE with an alias name, or as an ASSIGN command in which no alias is necessary.

Example

```
ASSIGN compkind     CODENAME(compkind).
```

To reassign the map value from metric to English:

```
ASSIGN map_l  map_l/25.4.
```

To use with the ROUND versus the sprint command:

```
ASSIGN elev_l   ROUND(elev_l * 3.28, -1).
```

To test for NULL:

```
ASSIGN fraggt10_l   if isnull(fraggt10_l) then 0 else
fraggt10_l.
```

In all instances, the original variable is recalculated but maintains the original variable name.

Names of variables may be any combination of letters, numbers, and the underscore character provided that the name starts with a letter and is not the same as one of the reserved words in the

language. In this document, reserved words are printed in sans-serif CAPITALS. Different scripts may use the same variable names, but the variables are independent, even if one script calls another via a DERIVE statement. There is one restriction on variable names for Properties that are used in interpretations. Evaluations take their input data from variables named "low", "rv", and "high" (or just "rv" if the property modality is RV). A Property called by an Evaluation can use other variables for intermediate results, but the final results must be in variables with these names.

Each variable is given a value by an expression or by a list of expressions separated by commas. An expression may be based on literals, columns from the input, or other variables. If a list is used, the listed values are combined into an array that has as many values as the sum of the number of values of the items in the list. On each cycle of the input, all variables are recalculated so that they appear in the DEFINE and ASSIGN statements. Variables are not explicitly typed, so the data type is determined by the result of the expression.

An initial value for a variable can also be specified in the DEFINE statement. A variable defined with an initial value and no expression is simply a constant; its value will not be changed. Only a single initial value can be specified, not a list of values.

An important use for an initial value is with an expression that contains the variable being defined. Consider the statement: `DEFINE list (list || name) INITIAL "Names: "`. This statement takes the column "name" from each input record and concatenates it to the variable "list", following the initial string `"Names: "`. If no initial value is defined with this type of expression, the variable starts out with a null value, which could produce undesirable results.

Storing Multiple Values in a Variable

A variable may hold a single value or multiple values. The number of values is called the variable's *dimension*. Multiple-valued variables are sometimes referred to as *arrays*. They can be created in several ways:

- With an AGGREGATE clause in an EXEC SQL statement.
- From a Property called by a DERIVE statement.
- From a list of values or expressions in a DEFINE statement.
- By using the APPEND operator.

Depending on the aggregations and other operations used, the variables in a report can end up with different dimensions. Some operators, such as LOOKUP and WTAVG, can cause a report to fail if the dimensions of their arguments are not the same. Therefore, attention must be paid to the way multiple-valued variables are processed.

Most of the operators used in expressions do not change the dimension of the data. If an operator uses two or more variables of different dimensions, the result generally has the largest dimension of the arguments. For example, multiplying a variable with values (1,2,3,4) by the single value 5 produces the multiple-valued result (5,10,15,20).

A query that finds no rows results in variables with a dimension of 0, which are typically treated the same way as null values. If all the arguments to an operator have dimension 0, the result also has dimension 0; but if there is a mixture of zero and non-zero dimensions, the result has the larger dimension. In the example above, multiplying the array (1,2,3,4) by a variable with no values would produce an array of four nulls.

Operators that do not follow these rules are noted in the individual descriptions below. Examples are the array operators like ARRAYSUM that reduce an array of values to a single value.

Expression Syntax

The following syntax rules define all the types of expressions that may be created.

expression ⇒ { literal | element | variable | arithmetic_expression | conditional_expression | boolean_expression | string_expression | regroup_expression | function | **(** expression **)** }

initialization ⇒ INITIAL literal

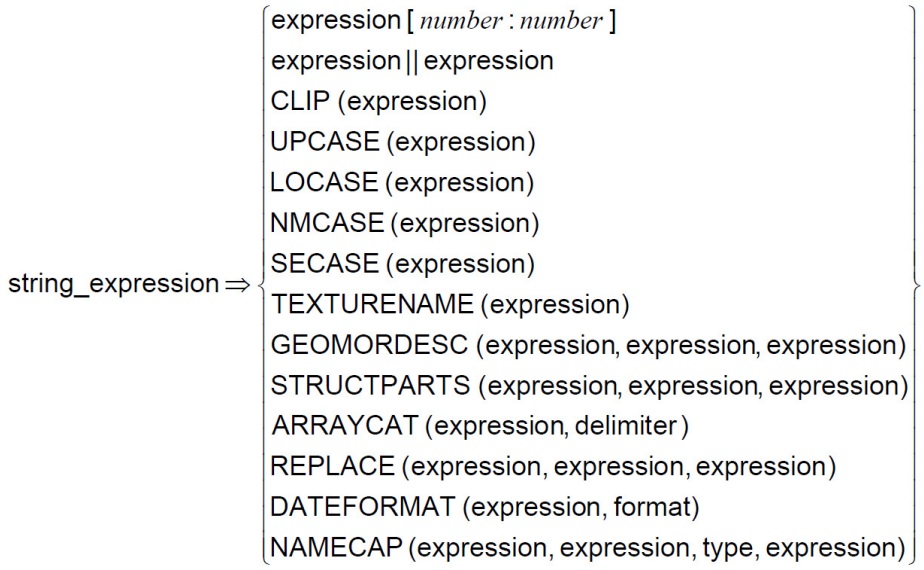
literal ⇒ { *number* | **"***string***"** }

arithmetic_expression ⇒ { expression { + | - | * | / | ** } expression | – expression }

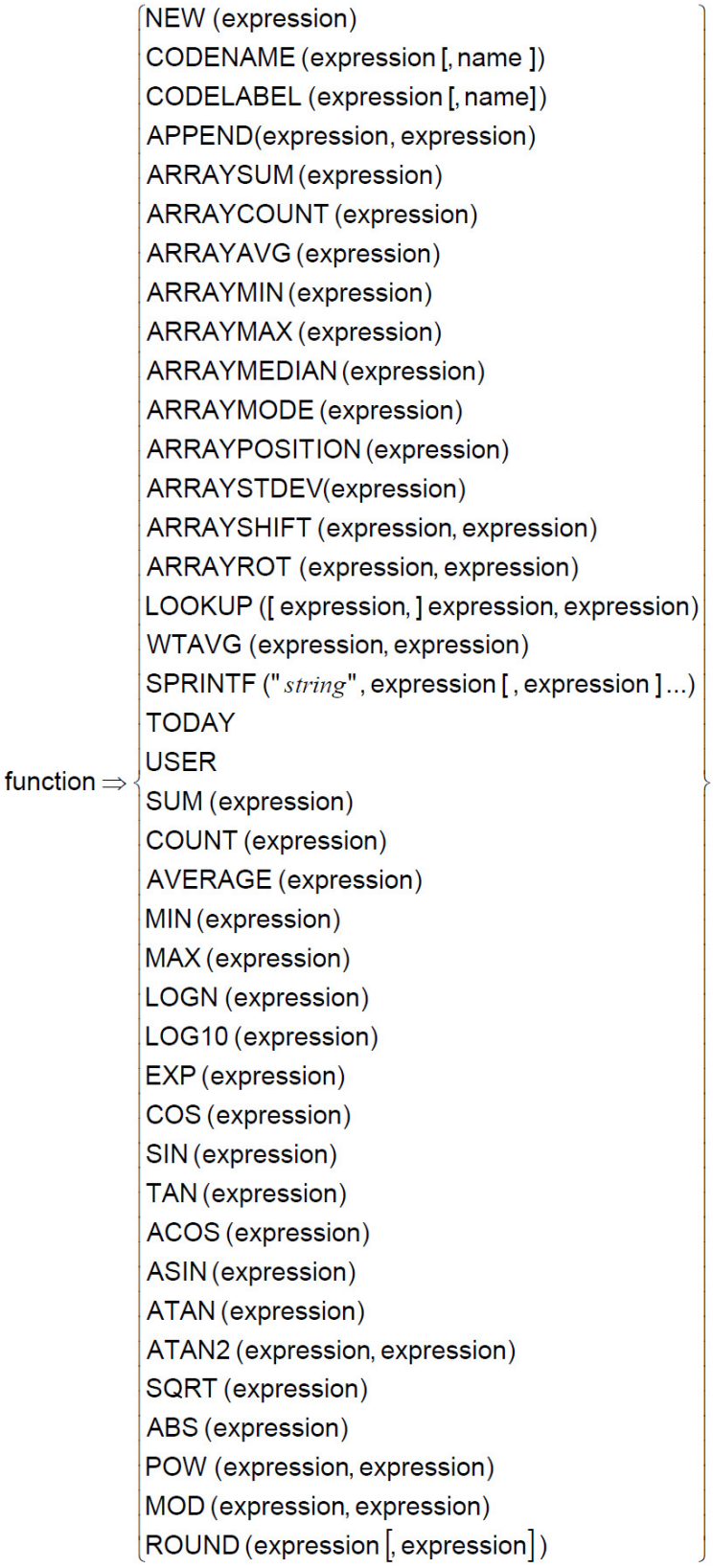
conditional_expression ⇒ { expression ? expression : expression | [IF] expression THEN expression ELSE expression }

boolean_expression ⇒ { comparison | ISNULL **(** expression **)** | NOT boolean_expression | ANY expression | ALL expression | **(** boolean_expression **)** | boolean_expression { AND | OR } boolean_expression }

comparison ⇒ expression { == | != | < | > | <= | >= | MATCHES | IMATCHES } expression



regroup_expression ⇒ REGROUP expression BY expression
AGGREGATE aggregate_function


function ⇒
NEW (expression)
CODENAME (expression [, name])
CODELABEL (expression [, name])
APPEND(expression, expression)
ARRAYSUM (expression)
ARRAYCOUNT (expression)
ARRAYAVG (expression)
ARRAYMIN (expression)
ARRAYMAX (expression)
ARRAYMEDIAN (expression)
ARRAYMODE (expression)
ARRAYPOSITION (expression)
ARRAYSTDEV(expression)
ARRAYSHIFT (expression, expression)
ARRAYROT (expression, expression)
LOOKUP ([expression,] expression, expression)
WTAVG (expression, expression)
SPRINTF (" string", expression [, expression]...)
TODAY
USER
SUM (expression)
COUNT (expression)
AVERAGE (expression)
MIN (expression)
MAX (expression)
LOGN (expression)
LOG10 (expression)
EXP (expression)
COS (expression)
SIN (expression)
TAN (expression)
ACOS (expression)
ASIN (expression)
ATAN (expression)
ATAN2 (expression, expression)
SQRT (expression)
ABS (expression)
POW (expression, expression)
MOD (expression, expression)
ROUND (expression [, expression])

Explanation of Expression Syntax

An expression can produce either a numeric value or a character-string value, depending on its contents. Numeric and character data can be mixed in expressions, and the data will be converted to the appropriate type if possible. If a conversion is not possible (such as trying to convert "abc" to a number), an error message will be produced.

Expressions are evaluated in order of operator precedence, where higher precedence operations are performed before lower precedence operations. For example, the arithmetic expression: `A + B * C` is evaluated as `A + (B * C)` because multiplication has higher precedence than addition. When two operators of equal precedence are next to each other, the one on the left is performed first. To make the order of evaluation explicit, put parentheses around the part that should be performed first, such as `(A + B) * C`.

The operator precedence from highest to lowest is:

- Functions
- Multiplication and exponentiation (*, / or **)
- Addition and concatenation (+, - or ||)
- Comparisons
- Boolean expressions
- Conditional expressions

Most of the expressions that involve arithmetic, Boolean, and comparison operators require little explanation. They work as expected and produce numeric results. The operator ** denotes exponentiation; the expression `A ** B` is equivalent to the function `POW(A, B)`.

Comparisons and Boolean expressions produce a 1 for True and a 0 for false. The MATCHES comparison works as in Informix (use in DEFINE statements). A variable can be compared with a pattern string containing wild card characters. The asterisk symbol * matches any string of characters, a question mark ? matches any single character, and square brackets [] enclose a list of characters to be matched. The IMATCHES comparison is the same as the MATCHES comparison but performs a case insensitive match. Use LIKE in the SQL.

If a null value is used in an expression, the result is normally null. However, in comparisons a null value is treated as less than any non-null value and two nulls are considered equal to each other. In Boolean expressions, a null is considered False. Invalid computations, such as division by zero, produce a null result. Special cases involving null values are noted individually.

String Expressions

String expressions allow for substring extraction, string concatenation, and case changes. They expect to operate on character type input and will convert the input to character if necessary.

Note that when a number is converted to a string, the number is expressed with 6 decimal places. To produce different formats for numbers, use the SPRINTF function. The results of the string expressions listed below are always character strings.

The use of the double pipe "||" allows for concatenating of fields, such as the Irrigated capability class and subclass.

```
DEFINE ilcCODELABEL(irrcapcl) || CODELABEL(irrcapscl).
DEFINE nilc          CODELABEL(nirrcapcl) ||
CODELABEL(nirrcapscl).
```

expression [n1:n2]

Returns a substring of the string expression, starting at position *n1* for a length of *n2* characters. The first character of the string is position 0. Note that this differs from the way substrings are defined in SQL queries.

Example: If variable A has the value "Sample", the expression `A[1:3]` returns the value "amp".

expression || expression

Concatenates two strings.

Example: The expression `"ABC" || "DEF"` produces the string "ABCDEF". If one expression in a concatenation is null, it is treated as the string ""; therefore, the result is not a null value unless both expressions are null.

Concatenating two fields with a dash in between:

```
DEFINE symname musym || "--" || muname.
```

Or concatenate a text string with a field:

```
DEFINE compsim "Description of  " || compname.
```

CLIP *(expression)*

Removes trailing blanks from a string. This is not normally necessary because NASIS removes trailing blanks when reading data from the database.

Example: The expression `CLIP("ABC ")` produces the string "ABC".

UPCASE (expression)

Converts a string to uppercase.

Example: The expression `UPCASE("ABc12")` produces the string "ABC12".

LOCASE (expression)

Converts a string to lowercase.

Example: The expression `LOCASE("ABc12")` produces the string "abc12".

NMCASE (expression)

Converts a string to "name" case: first letter of each word uppercase and the remainder lowercase.

Example: The expression `NMCASE("now is the time")` produces the string "Now Is The Time".

SECASE (expression)

Converts a string to "sentence" case: the first letter of the string is uppercase and the remainder is lowercase.

Example: The expression `SECASE("now is the time")` produces the string "Now is the time".

TEXTURENAME (expression)

Converts a set of texture codes to a special string format used in reports. The expression used by TEXTURENAME can have zero or more values, each of which is a string used as a code value for the NASIS data element "texture". This element can contain a mixture of codes for texture classes, modifiers, and terms used in lieu of texture. The codes are expanded and concatenated together, with commas as necessary, to produce a texture description as used in manuscript reports.

Example: If the variable T has two values, one of which is "SL", and the other is "SR-CL GR-SIL", the expression `TEXTURENAME(T)` produces a result with two values, the string "sandy loam", and the string "stratified clay loam to gravelly silt loam".

GEOMORDESC (expression, expression, expression)

Converts data from the component geomorphic description to a standard landform description string for use in reports. The three expressions used as input can be arrays, but all must have the same number of values. The first parameter is the feature name or names for a component, the second has the feature Id for each feature, and the third has the Exists-On reference for each feature. Where an Exists-On reference matches a feature ID, the two names are combined with the word "on". If two features have the same feature ID, the Exists-On reference is attached to both and they are output as separate strings. Other features that do not have an Exists-On relationship are output as

separate strings. The number of values in the result can be more or less than the number of values in the input expressions.

Example: Data for this operation would be obtained by joining the component geomorphic description table and the geomorphic feature table, such as:

```
EXEC SQL
SELECT geomorph_feat_name, geomorphic_feat_id,
exists_on_feature
FROM component
INNER JOIN component_geomorph_desc BY default
INNER JOIN REAL geomorph_feature BY default;
AGGREGATE COLUMN geomorph_feat_name NONE,
geomorphic_feat_id NONE, exists_on_feature NONE.
```

Assume this query produces the data shown in the following table:

geomorph_feat_name	geomorphic_feat_id	exists_on_feat
alluvial fan		
till plain	1	
pothole	2	1

The expression `GEOMORDESC(geomorph_feat_name, geomorphic_feat_id, exists_on_feat)` would produce a result with two values: “alluvial fan” and “pothole on till plain”.

STRUCTPARTS (expression, expression, expression)

Converts data from the Pedon Horizon Soil Structure table to a standard structure description string for use in reports. The parameters are used in the same manner as in the GEOMORDESC function above. The first parameter would be the type of structure, which is typically a string concatenated from structure_grade, structure_size, and structure_type. The second parameter is the row identifier, structure_id, and the third parameter is the reference column, structure_parts_to. The only difference between GEOMORDESC and STRUCTPARTS is that the latter uses the words “parting to” to separate linked structures instead of using “on”.

ARRAYCAT (expression, delimiter)

Concatenates the values in a multiple-valued variable or expression to produce a single-valued result. The first argument is a multiple-valued expression, and the second argument is a string to be used as a delimiter between the values. An empty string may be specified as the delimiter. If any values of the first argument are null, they and their

associated delimiters are skipped. The result has dimension 0 if the first argument has dimension 0, otherwise it has dimension 1.

Example: If the variable A has four values, "A1", "A2", Null, and "A4", then the expression `ARRAYCAT (A, "-")` would produce a single string: "A1-A2-A4".

REPLACE (expression, expression, expression)

Modifies character strings by replacing all occurrences of a sequence of characters with a replacement string. The first expression is the original character string to be modified. The second expression is a string to search for, and the third expression is a replacement string. Typically the second and third expressions will be single-valued, but the first expression can be multiple valued. The third expression can be an empty string, which causes all occurrences of the second string to be removed.

Example: If the variable A has the value, "This is a new test", the expression REPLACE (A, "new", "good") would produce: "This is a good test".

DATEFORMAT (expression, format)

Applies custom formatting to date/time data values in a DEFINE statement. The first expression is a variable containing dates as retrieved from the database. The format is a string in quotes that describes the desired date format. It follows the date-format rules for the Microsoft Net Framework, which is described in detail on their website. In general, the two kinds of date formats are (1) a single-letter format specifies one of the standard formats, and (2) a multiple-character string defines a custom format.

Standard formats are listed on the Microsoft help site and can be found by a search engine using the keywords "Microsoft net framework dateformat." Examples include:

"d"	Short date format	6/5/2009
"D"	Long date format	Friday, June 5, 2009
"g"	General date format (short time)	6/5/2009 1:45 PM
"G"	General date format (long time)	6/5/2009 1:45:30 PM

Examples of custom formats include:

"M/d/yy"	6/5/09
"MMMM d, yyyy"	June 5, 2009
"MM/dd/yyyy H:mm"	06/05/2009 13:45

Examples:

If the variable A has the value, "10/06/2008 14:10:05.1554", the expression `DATEFORMAT (A, "d")` would produce: "10/6/2008".

```
DATEDIFF (datepart, startdate, enddate)
```

NAMECAP (expression, expression, type, expression)

Applies standard capitalization rules for map unit names and component names. This is designed for use with the Calculations for map unit and component name capitalization. The first parameter is an array of component names that are exceptions to the standard. Normally, this array comes from the "name_exceptions" file distributed with NASIS. The second parameter is the array of component or map unit names to be standardized. The third parameter is "C" for components or "M" for map units. The fourth parameter is the map unit kind and is only required for map units.

In general, the capitalization standard is that the first letter of each component name is capitalized and everything else is in lowercase. The names in the exception list don't follow this rule. In addition, map unit names are arranged in a standard form depending on the map unit kind.

Example:

```
INPUT exceptions FILE "name_exceptions".
DEFINE stdnames
NAMECAP(exceptions, muname, "M", mukind).
```

Function Expressions

The following function expressions can use either character values or numeric values. These expressions produce results in the same type as the input, unless otherwise specified.

NEW (expression)

Returns True (1) if the value of the expression is different from the value it had in the previous cycle of the script, or returns False (0) if the value is the same.

Example: The expression `NEW (mapunit_symbol)` is True each time the mapunit symbol changes.

CODENAME (expression [, name])

Returns the code name for the code value given by *expression*, using the data dictionary domain of the element *name*. The *name* must be a data element *name* or its alias from an EXEC SQL statement. The value of the expression must be a number representing the

internal identifier for a code. This is the value normally returned by a query. If *expression* is the same as *name,* you do not have to specify it twice.

Example: If the variable *compkind* were returned from a query, the expression `CODENAME(compkind)` would produce a string normally displayed in NASIS for that data element, such as "series". Code names are generally in lowercase. The expression `CODENAME(val, compkind)`, where *val* is a variable from a DEFINE statement, would produce the code name for a compkind whose value is in the variable *val*.

CODELABEL (expression [, name])

Returns the code label for the code value given by *expression*, using the data dictionary domain of the element *name*. This operates the same as CODENAME. The code label is typically the same as the code name but is capitalized properly for use in reports.

Example: In the example above, the expression `CODELABEL(compkind)` would produce "Series".

APPEND (expression, expression)

Combines the values from two variables or expressions into a single variable. If the first expression has dimension n and the second expression has dimension m, the result of APPEND has dimension $n+m$ and contains all the values from the first expression followed by the values from the second. If an expression has dimension 0, it does not add anything to the result.

Example: If the variable A has three values, 1, 2, and NULL, and the variable B has the value 3, the expression `APPEND(A,B)` would have four values: 1, 2, NULL, 3.

ARRAYCOUNT (expression)

Counts the number of non-null values in a multiple-valued expression. The expression can operate on either a character or numeric argument, and it returns a single numeric value of zero or more.

Example: If the variable A has three values, 1, 7, and NULL, the expression `ARRAYCOUNT(A)` would produce the result 2.

ARRAYMIN (expression)

Computes the minimum of the values in a multiple-valued expression. The expression can operate on either a character or numeric argument, and it returns a single value of the same type as its argument. In this case, a null value is not considered to be smaller than a

non-null value. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0; otherwise, it has dimension 1.

Example: If the variable A has three values, 1, 2, and 3, then the expression `ARRAYMIN(A)` would produce the result 1.

ARRAYMAX (expression)

Computes the maximum of the values in a multiple-valued expression. The expression can operate on either a character or numeric argument, and it returns a single value of the same type as its argument. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0; otherwise, it has dimension 1.

Example: If the variable A has three values, "X", "Y", and "Z", the expression `ARRAYMAX(A)` would produce the result "Z".

ARRAYMEDIAN (expression)

Locates the median value in a multiple-valued expression, by sorting the non-null values and selecting the middle one. The expression can operate on either a character or numeric argument, but there is a slight difference in operation between the two. When there is an even number of values, there is no single middle value. The median reported, therefore, is the average of the two middle values for numeric data and the larger of the two values for character data. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0; otherwise, it has dimension 1.

Example: If the variable A has three values, "X", "Y", and "Z", the expression `ARRAYMEDIAN(A)` would produce the result "Y".

ARRAYMODE (expression)

Finds the modal value in a multiple-valued expression by counting the occurrences of each distinct value and returning the value that occurs most often. In case of a tie, the smallest value is returned. The expression can operate on either a character or numeric argument and returns a single value of the same type as its argument. The result is null only if all values of the array are null. The result has dimension 0 if the original expression has dimension 0; otherwise, it has dimension 1.

Example: If the variable A has four values, 2, 3, 5, and 3, the expression `ARRAYMODE(A)` would produce the result 3.

ARRAYSHIFT (expression, expression)

Shifts the values in the first argument, which is a multiple-valued variable, by the number of positions specified in the second argument, which has a single value. If the second argument (call it "n") is positive, the values are shifted "up", so that the value that was in position 1 moves to position n+1, and so on until the last n values are discarded. The first n array positions are assigned a null value. If the second argument is negative, the values are shifted in the opposite direction. The result has the same data type and number of values as the first argument.

Example: If the variable A has three values, 1, 2, and 3, the expression `ARRAYSHIFT(A, -1)` would produce a result with values 2, 3, and Null.

ARRAYPOSITION (expression)

Produces a new array of the same dimension as the argument and each position of the array having a sequential number starting with 1. This can be useful with the LOOKUP function to pick out a specific item from an array.

Example: If the variable A has three values ("x", "y", null), the expression `ARRAYPOSITION(A)` would produce a result with three values (1, 2, 3).

The following statement would find the third value (if there is one) in the array A:
`DEFINE third LOOKUP(3, ARRAYPOSITION(A), A).`

ARRAYROT (expression, expression)

Operates like ARRAYSHIFT but performs a rotation of the values in the first argument. Values shifted off one end of the array are moved onto the other end. If the number of positions shifted is greater than the number of values, the effect is to perform more than one rotation, or a rotation modulo the dimension.

Example: If the variable A has three values, 1, 2, and 3, the expression `ARRAYROT(A, -4)` would produce a result with three values, 2, 3, and 1.

LOOKUP ([expression,] expression, expression)

Selects values from an array based on an index or condition. If the LOOKUP has three parameters, the first expression is the key, which must be a single value, and the second expression is the index array. The key and the index must have the same type of data. If the key value is found in the index array, the value from the corresponding array position in the third expression is returned; otherwise, the result is null. If the LOOKUP has two parameters, the first expression is evaluated as an array of true or false values. If a value

is true, the corresponding array position in the second expression is returned. The result has the data type of the last expression.

There is a close relationship between the two forms of LOOKUP. The following two expressions produce the same result: LOOKUP(a,b,c) and LOOKUP(a==b,c). Use whichever form is easier to understand.

If there is more than one match or true value, the result has the values from all matching/true rows. It is therefore possible for the result to have more than one value. The last two expressions must be arrays of equal dimension. *A common error is to mismatch the dimensions of these two expressions due to differences in the way they are aggregated.*

Example: The variable *max_thickness* has a single number, the variable *horizon_thickness* has 6 numbers, and the variable *ph_r* has 6 numbers. The expression `LOOKUP (max_thickness, horizon_thickness, ph_r)` or `LOOKUP (horizon_thickness==max_thickness, ph_r)` would return the value of *ph_r* from the horizon whose *horizon_thickness* value equals the value of *max_thickness*.

COUNT (expression)

Maintains a running count of the occurrences of the expression. On each cycle of the script, the value of the expression is tested for a null, and if it's not null the counter's value is increased by one.

Example: A variable defined with the value `COUNT(musym)` could be printed at the end of a report to show the number of map units read (because musym can't be null).

MIN (expression)

Finds the smallest value of the expression. On each cycle of the script, the value of the expression is compared to an internal counter and replaces the counter's value if the expression is smaller. If a null value for the expression is encountered, the result of MIN becomes and remains null.

Internal counters for the MIN function cannot be reset.

Example: A variable defined with the value `MIN(elevation)` could be printed at the end of a report to show the minimum of elevation.

MAX (expression)

Finds the largest value of the expression. On each cycle of the script, the value of the expression is compared to an internal counter and replaces the counter's value if the

expression is greater. Null values are smaller than any non-null value, so the result is only null if all input values are null.

Internal counters for the MAX function cannot be reset.

Example: A variable defined with the value `MAX(elevation)` could be printed at the end of a report to show the maximum of elevation.

SPRINTF ("format", expression [, expression] ...)

Formats one or more expression values into a character string using the C function *sprintf* (same as the Prelude *sprintf*). The first argument is a format specification, which must have a single value, and the remaining arguments are expressions whose values are to be formatted. If any of the expressions are multiple valued, the result is also multiple valued and its dimension is that of the expression with the largest dimension.

It is the user's responsibility to ensure that the number and type of the expressions correspond to the format; no system checks are performed. Character data should use the %s formatting code, and numeric data should use the %f or %g formatting code.

Null values in the expressions produce an unusual result. The formatted value plus all characters of the format string up to the next % sign are skipped.

Examples:

```
DEFINE clay ISNULL(claytotal_l) OR ISNULL(claytotal_h) ? "
---" : sprintf("%3.f-%-2.f",claytotal_l,claytotal_h).
```

The variable *name* has one character value, "Bob". The variable *position* has two numeric values, 10 and 12. The expression SPRINTF ("%s:%.f", name, position) will produce a result containing two character values, "Bob:10" and "Bob:12".

Assigning a fixed number of decimals as in awc will be 4 places with 2 decimal places:

```
DEFINE awcISNULL(awc_l) OR ISNULL(awc_h) ? "   ---"
:sprintf("%4.2f-%-4.2f",awc_l,awc_h).
```

Some data fields can be NULL within the database, and a decision must be made to transform the data for use. DEFINE uses IF, THEN, ELSE to test for NULL values. (IF the unified field is null, THEN assign "NULL VALUE" ELSE code label the unified class.)

```
DEFINE un1ISNULL(unifiedcl) ? "NULL VALUE":
CODELABEL(unifiedcl).
```

USER

The user name from the data dictionary.

Example: If the person running NASIS has the login name "rose", the expression USER will return a single character value, "rose".

TODAY

The current date in mm/dd/yyyy format.

Example: 07/20/2018

STUFF function

The STUFF function inserts a string into another string. It deletes a specified length of characters in the first string at the start position and then inserts the second string into the first string at the start position.

```
STUFF ( character_expression , start , length ,
replaceWith_expression)
```

Example: This example switches the order of the user name in a project.

```
STUFF (username,1,CHARINDEX(',', username),'') + ' ' +
LEFT (username, CHARINDEX(',', username)-1) AS name
```

ISNULL function

Changes Null values to something else. Null values can be troublesome with some programs. The ISNULL function in the SELECT statement can change null values to zero for numeric fields and to a text value, such as "None", for text fields.

Examples:

This example changes Null project land category values with a dash

```
ISNULL (projectlandcategoryacres, '-') AS
projectlandcategoryacres
```

This example changes the horizon low depth to zero if the depth is null.

```
ISNULL (hzdept_l, 0) AS hzdept_l
```

This example changes a flooding frequency class from null to "None".

```
ISNULL (floodfreqcl, 'None') AS floodfreqcl
```

Numeric Functions

The following function expressions operate on numeric values and produce numeric results. If the input values are character strings, they are first converted to numbers.

ARRAYSUM (expression)

Computes the sum of the values in a multiple-valued expression. It expects a numeric argument and will try to convert character values to numbers. It returns a single numeric value. If individual values of the array are null, they are treated as zeroes. The result is null only if the array has no values. The result has dimension 0 if the original expression has dimension 0; otherwise, it has dimension 1.

Example: If the variable A has three values, 1, 2, and 3, the expression `ARRAYSUM(A)` would produce the result 6.

ARRAYAVG (expression)

Computes the average of the values in a multiple-valued expression. It expects a numeric argument and will try to convert character values to numbers. It returns a single numeric value. If individual values of the array are null, they are not counted in the average. The result is null if all values are null. The result has dimension 0 if the original expression has dimension 0; otherwise, it has dimension 1.

Example: If the variable A has three values, 1, 2, and 3, the expression `ARRAYAVG(A)` would produce the result 2.

ARRAYSTDEV (expression)

Computes the standard deviation of the values in a multiple-valued expression. It expects a numeric argument and will try to convert character values to numbers. It returns a single numeric value. If individual values of the array are null, they are not included in the computation. The result is null if all values are null. The result has dimension 0 if the original expression has dimension 0; otherwise, it has dimension 1.

Example: If the variable A has three values, 1, 2, and 3, the expression `ARRAYSTDEV(A)` would produce the result 1.

WTAVG (expression, expression)

Computes the sum of the first expression's values after multiplying each by a weighting factor, which is taken from the corresponding value of the second expression, then divides the result by the sum of the weights. The two expressions must be arrays that have the same dimension. Individual null values are ignored in computing the average.

The result is null if all the individual values are null. The result has dimension 0 if the original expressions have dimension 0; otherwise, it has dimension 1.

Example: The variable *comppct_r* has 3 values (40, 30, 20) and the variable *elev_r* has three values (1000, 1200, 900). The expression `WTAVG (elevation, comppct_r)` would produce the value 1044.44, which is the average of the *elevation* values, weighted by the *comp_pct* values, or (1000*40 + 1200*30 + 900*20) / (40 + 30 + 20).

SUM (expression)

Computes a running total of the value of the expression. On each cycle of the script, the value of the expression is added to an internal counter. The result of the function is the value of that counter at each cycle. If a null value for the expression is encountered, the result of SUM becomes and remains null.

Internal counters for the SUM function cannot be reset. If you want to compute subtotals, use the ASSIGN statement to add the value of the expression to a defined variable rather than an internal counter. If you use the ASSIGN statement, a conditional expression can be used to reset the variable's value to 0 at the correct time.

Example: A variable defined with the value `SUM(acres)` could be printed at the end of a report to show the total of acres.

AVERAGE (expression)

Computes a running average of the value of the expression. On each cycle of the script, the value of the expression is added to an internal counter, and the result is divided by the number of values processed. If a null value for the expression is encountered, the result of AVERAGE becomes and remains null.

Internal counters for the AVERAGE function cannot be reset.

Example: A variable defined with the value `AVERAGE(elev_r)` could be printed at the end of a report to show the average of elevation.

LOGN (expression)

Computes the natural logarithm of the expression.

Example: The expression `LOGN(10)` produces the value 2.302585.

LOG10 (expression)

Computes the base 10 logarithm of the expression.

Example: The expression `LOG10(10)` produces the value 1.

EXP (expression)

Computes the exponential (e^x) of the expression.

Example: The expression `EXP(1)` produces the value of e, 2.718282.

COS (expression)

Computes the cosine of the expression interpreted as an angle in radians.

Example: The expression `COS(0)` produces the value 1.

SIN (expression)

Computes the sine of the expression interpreted as an angle in radians.

Example: The expression `SIN(0)` produces the value 0.

TAN (expression)

Computes the tangent of the expression interpreted as an angle in radians.

Example: The expression `TAN(0)` produces the value 0.

ACOS (expression)

Computes the arccosine of the expression, returning an angle in radians.

Example: The expression `ACOS(0)` produces the value of π/2, 1.570796.

ASIN (expression)

Computes the arcsine of the expression, returning an angle in radians.

Example: The expression `ASIN(1)` produces the value of π/2, 1.570796.

ATAN (expression)

Computes the arctangent of the expression, returning an angle in radians.

Example: The expression `ATAN(1)` produces the value of π/4, 0.785398.

ATAN2 (expression, expression)

Computes the angular component θ of the polar coordinates *(r, θ)* that are equivalent to the rectangular coordinates *(x, y)* given by the two expressions. This is the same as `ATAN(y / x)`.

Example: The expression `ATAN2(5, 5)` produces the value of π/2, 1.570796.

SQRT (expression)

Computes the square root of the expression. Returns a null value if the expression is negative.

Example: The expression `SQRT(2)` produces the value 1.414214.

ABS (expression)

Computes the absolute value of the expression.

Example: The expression `ABS(-10)` produces the value 10.

POW (expression, expression)

Computes the value of the first expression raised to the power of the second expression.

Example: The expression `POW(2, 5)` produces the value 32.

MOD (expression, expression)

Computes the remainder after dividing the first expression by the second expression.

Example: The expression `MOD(5, 2)` produces the value 1.

ROUND (expression [, expression])

Rounds off the value of the first expression to the number of decimal places specified by the second expression. If the second expression is not specified, it is assumed to be zero, which means round off to the nearest whole number. When the second expression is a positive number, it specifies the number of places to the right of the decimal point to be preserved. If negative, it means round to the specified number of places to the left of the decimal point, as illustrated in the examples.

Examples

`ROUND (15.751, 1)` produces 15.8
`ROUND (15.751)` produces 16
`ROUND (15.751, -1)` produces 20

REGROUP Expression

The REGROUP expression is used to perform secondary aggregation of data. It operates a little like the AGGREGATE option in a query and can be used to perform a second level of aggregation when dealing with a complex data structure. It uses two expressions, which must be arrays of the same dimension. In the expression "REGROUP array BY array …", the second array (the "BY" array) is used as a key for grouping the values from the first array (the data array). The result is a new array whose dimension is the number of unique values in the "BY" array. The values in the result are aggregates derived from each group of rows in the data array that have the same key value.

The REGROUP statement is a secondary aggregation. REGROUP reduces each variable to one value. REGROUP is part of the aggregation clause and allows for re-aggregation of your data after the initial aggregation. The unified (all horizon data) is aggregated using NONE so that the horizon level data is not aggregated to the component name. It is regrouped below to have it aggregate at the horizon level instead of the component level.

The aggregation function determines how these aggregates are produced. The types of aggregation are the same as the query AGGREGATE option, except that NONE and UNIQUE are not applicable in REGROUP because each position of the result array can have only one value. The valid aggregations types are:

- SUM computes the sum of the values in each group.
- AVERAGE computes the average of the values in each group.
- FIRST selects the value from the first row of the group.
- LAST selects the value from the last row of the group.
- MIN selects the smallest of the values in each group.
- MAX selects the largest of the values in each group.
- LIST concatenates the values (converted to character strings if numeric) into a single string with a delimiter between each value. If a quoted string is specified after the word LIST, that string is the delimiter; otherwise, a comma and space are placed between each value.
    - Example: Aggregate column hzname list ",".
    - This produces a single array similar to: A,E,Bt,C

Some additional rules on the REGROUP expression:

The "BY" array does not have to be sorted. REGROUP always collects together all data values for each unique key value. However, the choice of value for FIRST or LAST is affected by the order of values in the data array.

Nulls in the data array are ignored during aggregation except for FIRST and LAST, which preserve a null if it is the first or last value found. If all data values for some key value are null, the corresponding result value is null.

A null in the "BY" array is a valid key value and produces a corresponding value in the result, aggregating all null key values together.

Example: These examples use the arrays A and B as inputs:

A	B
George	4
Abe	4
Sue	5
Sam	8
Mary	8
William	8

The arrays C and D are produced by the statements:

```
DEFINE C REGROUP A BY B AGGREGATE FIRST.
DEFINE D   REGROUP A BY B AGGREGATE LIST "-".
```

C	D
George	George-Abe
Sue	Sue
Sam	Sam-Mary-William

Example

```
BASE TABLE component.

EXEC SQL

SELECT compname, slope_r, hzname, hzdept_r, hzdepb_r,
claytotal_r, unifiedcl, chunified.rvindicator

FROM component

INNER JOIN chorizon BY DEFAULT

INNER JOIN chunified BY DEFAULT

WHERE chunified.rvindicator = 1;

SORT by compname, hzdept_r
```

```
AGGREGATE ROWS compname

COLUMN hzname none, hzdept_r none, hzdepb_r none,
claytotal_r none, unifiedcl none.
```

This query creates the first aggregation on the component name so each component has all of its various horizons and clays and unified texture. Many components and all of their data.

```
ASSIGN unifiedcl REGROUP codename (unifiedcl) by hzname
aggregate list ", ".

ASSIGN hzdept_r REGROUP hzdept_r by hzname aggregate first.

ASSIGN hzdepb_r REGROUP hzdepb_r by hzname aggregate first.

ASSIGN claytotal_r REGROUP claytotal_r by hzname aggregate
first.

ASSIGN hzname    REGROUP hzname by hzname aggregate first.
```

Note: Always regroup hzname last.

Report Syntax

DERIVE

Syntax

DERIVE derive_list USING property_call .

derive_list ⇒ variable [FROM identifier] [, variable [FROM identifier]] ...

property_call ⇒ ["*site_name*" :]"*property_name*"

[(argument [, argument] ...)]

argument ⇒ { variable | element | literal }

Used In

Report, Property, Calculation

Example

```
DERIVE thickness FROM layer_thickness

USING "NSSC_Pangaea":"LAYER THICKNESS" (0, bottom).
```

The DERIVE statement invokes a property script to produce values for one or more variables. Each name listed after the keyword DERIVE becomes a local variable in the script where it occurs. It is assigned the value of the variable in the property script whose name follows the keyword FROM. If the names before and after FROM are the same, the FROM phrase may be omitted. The property must have the same base table as the calling script, and the scripts are automatically synchronized to return values for the current row of the base table.

The name of the property must be in quotes and must match the property name in the Property table exactly, including case and punctuation. The NASIS site name is optional but should be placed before the property name to ensure that the name is unique.

If the property script has an ACCEPT statement, a list of arguments must be given after the property name. The order of the arguments in the DERIVE statement must correspond to the order of the input variables in the ACCEPT statement.

The arguments can be input column names, variables, or numeric or character constants in the calling script. However, recall that DERIVE statements are always executed before DEFINE statements. If an argument is a variable that is computed in a DEFINE statement, its value is whatever is left over from the previous script cycle, even if the DEFINE appears in the script before the DERIVE. For this reason, arguments for DERIVE should be from an ACCEPT, an EXEC SQL, or constants.

EXEC SQL

Syntax

EXEC SQL sql-select [sort_specification] [aggregation] .

sql-select ⇒ SELECT [DISTINCT] [TOP n] input_column [, input_column] ...

FROM table_spec [, table_spec] ...

[WHERE where_condition [{AND | OR} where_condition] ...]

[*SQL GROUP BY clause*] [*SQL HAVING clause*]

[{ *SQL ORDER BY clause* | *SQL INTO TEMP clause* }] ;

input_column ⇒ { element [[AS] alias] | expression [AS] alias }

element ⇒ [{ *TablePhysicalName* | *TableLogicalName* | alias } .] { *ColumnPhysicalName* | *ColumnLogicalName* }

alias ⇒ *name*

table_spec ⇒ [{ EDIT | REAL }] [OUTER] { *tbl_imp_nm* | *tbl_nm* } [alias] [join]

join ⇒ [{ INNER | {LEFT | RIGHT} OUTER | CROSS }] JOIN table_spec { ON *SQL WHERE condition* | BY *relationship_name* }

where_condition ⇒ { *SQL WHERE condition* | JOIN table TO table [BY *relationship_name*] }

Used In

Report, Property, Calculation

Example

```
EXEC SQL
SELECT musym, muname
FROM Mapunit;.
```

An EXEC SQL statement defines a database query that supplies input to the report engine. Any database columns or expressions listed in the SELECT clause of the query may be used as variables in the rest of the script. A script almost always has a query, the exceptions being reports that get all their data from files, parameters, or derived properties. The primary purpose of EXEC SQL is to specify which data elements are necessary for the report.

The EXEC SQL statement is a variation of a standard SQL Select statement. It performs the same basic function but has additional capabilities to make report writing easier. The additional capabilities include:

- Use of NASIS logical column names as well as database column names;
- Simplified syntax to specify join conditions;
- Extended sort types, such as case insensitive and symbol sort; and
- More powerful GROUP BY features in the AGGREGATE clause, including independent aggregation by column and crosstab formatting.

An SQL Select statement begins with a SELECT clause. It has a list of database columns or expressions, following normal SQL syntax, and each column must have a unique name. If expressions are used in the select list, an alias must be used with the expression to provide a unique name. In addition to allowing most standard SQL expressions, NASIS permits the functions CODENAME, CODELABEL, CODESEQ and CODEVAL with data elements that are stored as codes. These functions cause the query to return the name, label, sequence, or internal value for a code. If none of these functions is used, the query returns the internal value.

The word DISTINCT following the word SELECT removes duplicate rows from the query results. If the combination of the values of all items listed in the SELECT clause is duplicated, only one occurrence will be produced. (In prior versions of NASIS, the word UNIQUE could be used instead of DISTINCT, but this is no longer allowed in SQL). DISTINCT finds the unique values for the total row not just the first field in the list. Use EXISTS in a subquery to find unique values for specific fields.

The term IS NULL preceding a field and a replacement term in the SELECT clause changes all null values to the new term.

Example: `SELECT IS NULL(slope_r, 0) AS slope`

This will change all null values for slope_r to zero.

The phrase TOP *n* following the word SELECT is a SQL feature that allows you to specify the maximum number of records to be returned from a query. The records are sorted on the columns specified in the ORDER BY clause, then up to "*n*" of them are used as report input. This is handy to use while testing a report.

The FROM clause specifies all the tables used in the query and may also specify aliases and joins. Table names in a FROM clause must be defined in the NASIS data dictionary or in an INTO TEMP clause of a prior query. Aliases may also be used with table names. They provide a shorter name or make the name unique. Short one or two letter alias names are indexed faster than longer names or the original field name.

A CVIR script is designed to search either the selected set or the full local database, depending on the type of script. Normally, reports search only the selected set, while calculations and properties search the whole local database. The keyword EDIT or REAL in the FROM clause overrides the table search behavior on a table-by-table basis. If the keyword EDIT is used, only the selected set is read; and if REAL is used, the whole local database is read. If a report is run on the national database, the EDIT or REAL option is ignored and the whole national database is read. EDIT is the default search behavior.

The FROM clause can also contain specifications for joining tables using current SQL syntax (as well as the older syntax based on Informix). The newer style of join puts all the join conditions in the FROM clause, as in the following examples.

```
FROM datamapunit
INNER JOIN component BY DEFAULT

FROM datamapunit
LEFT OUTER JOIN component ON
datamapunit.dmuiid=component.dmuiidref
```

For a complete discussion about this kind of join, you'll need an SQL manual or class. Some of the key points are:

- When the join conditions begin with ON, standard SQL syntax applies. You must specify the exact columns to be matched in each of the tables. You can also add more conditions beyond just the key columns. The report in NASIS "!Join statements" in the Documentation folder has all of the joins for all the tables in NASIS that can be copied and pasted into a SQL script.

- When using the BY condition, you specify a relationship name defined in the NASIS data dictionary. In most cases the relationship name is "default". If more than one relationship exists between a pair of tables, you must use the correct name. The Info page for a table in NASIS lists the relationship names.

Advanced Note

There is an important case in which a condition must be in the WHERE clause. This condition is related to a difference between Informix and SQL Server. When an outer join is used, you need to apply additional selection criteria to the outer (not required) table. These criteria operate differently in the FROM clause than in the WHERE clause.

Here are two examples:

```
A. FROM component
   LEFT OUTER JOIN comonth BY DEFAULT
   AND 'month' = 'jan'

B. FROM component
   LEFT OUTER JOIN comonth BY DEFAULT
   WHERE 'month' = 'jan'
```

Example query A produces a row for a Component that does not have a January in the Component Month table, but example query B will not. The reason is that the WHERE conditions are applied to the result of the join. In query B, each Component will be joined up with its Component Month rows (if any), and then only those with January will be selected. In query A, the selection of January records occurs during the join process. If there are none, the outer join applies and the Component is included in the output even though there is no matching Component Month.

Following are four more examples showing a LEFT OUTER JOIN in which the conditions are in different places in the script.

1) When both conditions are in the FROM clause, the query returns 1,483,094 rows of data.

```
EXEC SQL
SELECT COUNT(coiid) AS compcount
FROM component
LEFT OUTER JOIN comonth ON
comonth.coiidref=component.coiid AND month='jan' AND
compname='relfe';.
```

2) When the right table condition is in the FROM clause and the left table condition is in the WHERE clause, the query returns 92 rows.

```
EXEC SQL
SELECT COUNT(coiid) AS compcount
FROM component
LEFT OUTER JOIN comonth ON
comonth.coiidref=component.coiid AND month='jan' WHERE
compname='relfe';.
```

3) When both conditions are in the WHERE clause, the query returns 70 rows of data.

```
EXEC SQL
SELECT count(coiid) as compcount
FROM component
LEFT OUTER JOIN comonth ON
comonth.coiidref=component.coiid
WHERE month='jan' AND compname='relfe';.
```

4) When the left table condition is in the FROM clause and the right table condition in the WHERE clause, the query returns 70 rows.

```
EXEC SQL
SELECT count(coiid) AS compcount
FROM component
LEFT OUTER JOIN comonth ON
comonth.coiidref=component.coiid AND compname='relfe'
WHERE month='jan';.
```

If you don't use the type of join shown above, the WHERE clause may use the "JOIN *table* TO *table*" syntax as in NASIS 5. The two tables in a JOIN condition must have a relationship recorded in the data dictionary. In this form, the BY phrase can be omitted if only one relationship exists between the two tables. You can also use the word OUTER in the FROM clause in combination with this type of join specification. NASIS will internally convert it to the new style join.

Subqueries are also allowed in the WHERE clause following SQL syntax with the extensions just described. It is permissible to use a JOIN condition between a table listed in the main query and a table listed in the subquery, which is a convenient way to create a coordinated subquery. Refer to SQL references for more information about subqueries. This is an advanced query topic.

The expressions in the WHERE clause may use variables defined in a PARAMETER statement, an ACCEPT statement, or a prior query. The CVIR engine plugs in the values of such variables at the time the query is executed. If the variable is preceded by a dollar sign, such as $name, the

SQL query becomes a "parameterized query." A parameterized query does not use the automatic query coordination specified by the BASE TABLE statement. The selection is instead controlled by the parameter value. With this technique, a report can have a mix of queries that have different aggregation levels.

Examples can be found in the map unit description reports.

Following the WHERE clause, the GROUP BY, HAVING, and ORDER BY clauses can be used with the normal SQL syntax. The INTO TEMP clause may also be used (provided that ORDER BY is not used) to direct the results of the query into a temporary database table. Subsequent queries in the same script or in subreport scripts can read from the temporary table as if it were a normal NASIS table. The column names in the temporary table are the column names (or aliases) from the SELECT clause. A query with an INTO TEMP clause should not be the only query in a report because it does not return any data that the report can use.

NASIS tables and columns may be called by either the logical name or the physical name, but they must use the same name wherever referenced. A column name can be used alone if it is unique; otherwise, the table name must be given also. If an alias is used for a column in a SELECT clause, that alias must be used everywhere instead of the column name. If the column is modal, the suffix (such as _l or _h) must be included in the name. The value from each column is converted into either a character string or a floating-point number for use in later calculations.

Important: Numeric data elements, such as Integer, Decimal, and Float, and Code elements, are converted to floating point, and everything else, including dates, is converted to character strings.

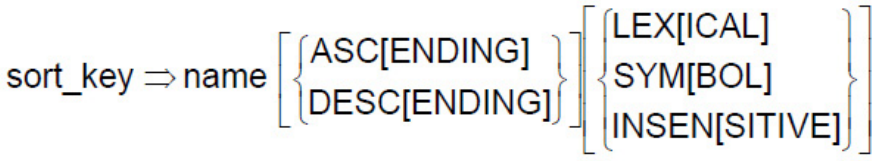
A semicolon is required to end the SQL portion of the EXEC SQL statement. Optional sort and aggregation clauses may follow the semicolon, and the whole statement is ended with a period. If neither a sort nor an aggregation clause is used, both a semicolon and a period are still required.

A script may contain more than one query to collect data from different hierarchic paths in the database. These types of data commonly cannot be retrieved in a single query without creating undesirable cross products. By using separate queries and aggregating the results, the data can be "de-normalized" so that data from separate paths appear as if they were repeating groups in the base table.

EXEC SQL: Sort Specification

Syntax

sort_specification ⇒ SORT [BY] sort_key [, sort_key] ...



Example

```
EXEC SQL
SELECT muname mname, nationalmusym, dmuiid
FROM mapunit
INNER JOIN correlation BY DEFAULT
INNER JOIN datamapunit BY DEFAULT
WHERE repdmu = 1 AND muiid in ($muiid);
SORT BY mname, dmuiid.
```

SORT is an optional clause that may be added to a query to direct the CVIR engine to sort the records. Either ORDER BY (which causes the database engine to do the sorting) or SORT may be used; the SORT takes precedence. The SORT clause is slower but provides more options than ORDER BY. The sort key names in the SORT clause must be column or alias names used in the SELECT clause (column numbers are no longer allowed). The direction of sorting (ascending or descending) can be specified for each sort key. The default is ascending. The type of sort can also be specified as lexical (like a dictionary), symbol (used for symbols containing both letters and numbers), or insensitive (ignore uppercase and lowercase distinctions). The default sort type is the one specified in the data dictionary for the column. The sort order and type keywords may be abbreviated as shown.

The difference between SORT and ORDER BY is important when the TOP n condition is used in the SELECT clause. ORDER BY, because it is performed by the database engine, happens before the "top n" records are selected and SORT happens after. It could even be useful to specify different columns in ORDER BY and SORT, because the first controls which records appear and the second controls the order in which they print.

EXEC SQL: Aggregation Specification

Syntax

aggregation ⇒ AGGREGATE [ROWS [BY] identifier [, identifier] ...]
[COLUMN identifier [aggregate_function]
[, identifier [aggregate_function]] ...]
[CROSSTAB [BY] identifier [value_specification]
[LABELS "*string*" [, "*string*"] ...]
CELLS identifier [, identifier] ...] .

identifier ⇒ { element | alias }

aggregate_function ⇒ { SUM | AVERAGE | FIRST | LAST | MIN | MAX | LIST ["string"] | NONE | UNIQUE } [GLOBAL]

value_specification ⇒ { VALUE[S] (field_value [, field_value] ...) | INTERVAL[S] (field_value [, field_value] ...) }

field_value ⇒ literal

Example

```
EXEC SQL
SELECT musym, muname, areaname, lmuaoverlap.areaovacres
acres
FROM area
INNER JOIN laoverlap BY DEFAULT
INNER JOIN lmuaoverlap BY DEFAULT
INNER JOIN lmapunit BY DEFAULT
INNER JOIN mapunit BY DEFAULT;
SORT BY musym SYMBOL, areaname
AGGREGATE ROWS BY musym
```

```
COLUMN muname UNIQUE, acres SUM
CROSSTAB areaname CELLS acres.
```

The aggregation clause specifies how the input records are to be grouped and what to do with the data in each group. The first query in a report (the "primary" query) can use the ROWS option to control how its records are grouped, but the remaining queries (the "secondary" queries) cannot have a ROWS option and instead use a simple global aggregation. This difference is significant in the following explanations.

A primary query without a ROWS option uses no aggregation, meaning that data records are used one at a time exactly as they come from the query. When the ROWS option is used, each unique combination of values in the ROWS columns starts a new cycle of report processing. The query must be sorted on the columns listed after ROWS, and it may produce more than one record for each combination of the ROWS columns. When a group has multiple records, the input values in each column are combined according to the aggregation rules and produce single values or arrays that can be used in further calculations or report output.

The behavior of row aggregation is illustrated in the following example. Suppose a query includes specifications to SORT BY musym AGGREGATE ROWS BY musym. Rows with the same musym define the report cycle. Each group of rows with the same musym is one cycle; thus, this example has eight rows but only three cycles.

Musym	compname	comppct_r
12A	Hamerly	80
12A	Vallers	15
12A	Hamre	5
26B	Windsor	90
26B	Deerfield	10
130C	Dacono	85
130C	Satanta	10
130C	Altvan	5

Aggregation rules for each column can be specified after the keyword COLUMN. The default aggregation is UNIQUE for columns that have no aggregation specified. This means that when the value in a column is the same for every row in a report cycle, only one value is returned for that column. If more than one value occurs in a cycle, an array is formed to return values for the column. Each distinct, non-null value is placed in a separate position of the array. The number of positions in the array (the dimension) can vary from one cycle to the next and from one column to another within a cycle.

The aggregation function NONE is similar to UNIQUE, except that NONE does not eliminate duplicate or null values. If there is more than one input row in a cycle, the value from each row is placed in a separate array position. For each cycle, every column with aggregated by NONE will have the same dimension, and the values will be in the order of the input records.

The other aggregation functions are used to reduce multiple values for a column to a single value. The aggregations have no effect when only one record occurs in a cycle. The types of aggregation are:

- SUM.—Computes the sum of the column's values.
- AVERAGE.—Computes the average of the column's values.
- FIRST.—Selects the value from the first record of the group (useful only if the input is sorted on this column).
- LAST.—Selects the value from the last record of the group.
- MIN.—Selects the smallest value in the column.
- MAX.—Selects the largest value in the column.
- LIST.—Concatenates the values (converted to character strings if numeric) into a single string that has a delimiter between each value. If a quoted string is specified after the word LIST, that string is the delimiter; otherwise, a comma and a space are placed between each value.

Given the example in the table above, suppose the query includes specifications to `AGGREGATE ROWS BY musym COLUMN compname LIST, comppct_r SUM`. Because aggregation for musym is not specified, the default aggregation UNIQUE is applied to that column to produce the following results. Note that the values in each column have been reduced to a single value for each cycle.

musym	compname	comppct_r
12A	Hamerly, Vallers, Hamre	100
26B	Windsor, Deerfield	100
130C	Dacono, Satanta, Altvan	100

Important:

The keyword GLOBAL may be used after the aggregation type for a column. This causes that column to be aggregated over the entire set of input data rather than over one cycle. The values for that column remain constant for the whole report. One use for global aggregation is to find data for report headings. If the first input cycle is missing some data needed in a heading, a global aggregation can find either the first occurrence or all unique occurrences of the data before the report processing begins.

The aggregation of secondary queries is like global aggregation. The ROWS option is not allowed, and the whole set of records that the query produces in each report cycle is aggregated together. Column aggregation rules can be specified for the columns of a secondary query if the default option of UNIQUE aggregation is not wanted.

A secondary query normally is automatically coordinated with the primary query via the Base Table. A hidden WHERE condition is applied to a secondary query so that it produces only the rows that match the current base table row. If the secondary query is "parameterized," meaning that it has references to variables in $name format, the automatic coordination is not used. Instead the parameter values are inserted into the query each cycle to control the selection of records.

CROSSTABS

CROSSTAB is a special type of aggregation that assigns values to positions in an array based on the value of a controlling column. It requires a CROSSTAB column and one or more CELLS columns. These columns become arrays, but their dimension is not determined by the number of input rows in a cycle but rather by the number of values for the crosstab. This dimension is constant for the entire query. The crosstab values are defined by the VALUES list, the INTERVALS list, or by default. The default is to use all the unique values found in the input for the crosstab column.

For each cycle of the input, CROSSTAB first sets to nulls the arrays of values for the CELLS columns. Then, for each input record, the value in the CROSSTAB column is examined. If the value is in the VALUES list, in default list, or falls within one of the ranges in the INTERVALS list, the position of the value in the list is noted. For each of the columns in the CELLS list, the value from the input record is placed in that position of the column's array.

Within a cycle, the value of the crosstab column may repeat. If so, only one value can be stored in an array position for a cell; therefore, the cell's aggregation function is applied. If a cell has no aggregation, a data row is returned for each unique value. In each such data row, all aggregated columns have constant values. The operation of crosstab can be illustrated using the following example data.

musym	muname	areaname	acres
10A	Alpha loam, 0 to 3	X	100
10A	Alpha loam, 0 to 3	X	200
10A	Alpha loam, 0 to 3	Y	300
10A	Alpha loam, 0 to 3	Z	400
10A	Alpha loam, 0 to 3	Z	500
10B	Alpha loam, 3 to 6	X	600
10B	Alpha loam, 3 to 6	Y	700
10B	Alpha loam, 3 to 6	Y	800

This table shows a small sample of input data from the example query above. The first case shows the results of a crosstab without aggregation of the crosstab cells:

```
AGGREGATE ROWS musym COLUMN muname UNIQUE CROSSTAB areaname
CELLS acres
```

musym	muname	areaname			acres		
10A	Alpha loam, 0 to 3	X	Y	Z	100	300	400
10A	Alpha loam, 0 to 3	X	Y	Z	200		500
10B	Alpha loam, 3 to 6	X	Y	Z	600	700	
10B	Alpha loam, 3 to 6	X	Y	Z		800	

In this example, the column “musym” controls row aggregation. Column “muname” has the UNIQUE aggregation, so it maintains the values that correspond to each value of “musym”. Note that if “muname” does not repeat at the same frequency as “musym”, it becomes an array.

The columns “areaname” and “acres” become arrays of three positions each because the crosstab column (areaname) has three distinct values in the input sample. The values placed in “areaname” are constant, namely the column grouping values “X”, “Y”, and “Z”. The cell column (acres) contains the acreage values for the corresponding position of “areaname”. Because there are multiple acreage values for each area in this example, the result has two rows for each symbol.

By adding an aggregation function to the “acres” columns, the crosstab produces just one row for each cycle defined by the ROWS condition, as shown in the following example.

```
AGGREGATE ROWS musym COLUMN muname UNIQUE, acres SUM
CROSSTAB BY areaname CELLS acres
```

musym	muname	areaname			acres		
10A	Alpha loam, 0 to 3	X	Y	Z	300	300	900
10B	Alpha loam, 3 to 6	X	Y	Z	600	1500	

When INTERVALS are used for a crosstab, the list of field values must be numbers in an increasing order. The number of intervals is one more than the number of values. If the intervals are specified as `CROSSTAB BY x INTERVALS (n1, n2, n3)`, the crosstab will place the cell data into one of 4 array positions based on the value of the variable x.

x <= n1 n1 < x <= n2 n2 < x <= n3 n3 < x

The LABELS specification can specify column headings for a report, which would otherwise be the field values for the CROSSTAB BY column. See the discussion about array specifications and column specifications under the SECTION statement for more information about formatting and printing cross tabulated data.

FONT

Syntax

FONT “font name”.

Used In

Report

Example

```
FONT “Courier”.
```

This statement has no function in NASIS 7 and is ignored. NASIS does not control the font used to display a text-style report when the report is opened in an application of the user’s choice. In an application like Notepad, it is recommended that you use a Courier font so that the output will look like it did in NASIS 5. For HTML-style reports the font is controlled by the style sheet or by attributes of the HTML tags.

```
ELEMENT ‘p’ ATTRIB (‘style’, font=Courier)

ELEMENT ‘p’ ATTRIB ('style', 'text-align:center;font-
family: Arial;font-size: 12')
```

HEADER and FOOTER

Syntax

```
HEADER [ INITIAL ]
        line-specification ...
END HEADER .
FOOTER [ FINAL ]
        line-specification ...
END FOOTER .
```

Used In

Report (text style only)

Example

```
HEADER
AT CENTER "Sample Report".
SKIP 2 LINES.
END HEADER.
```

Defines the headers and footers for the report. There are four types of header and footer statements, and a report may contain no more than one of each type. All are optional. The default for HEADER and FOOTER is to print nothing. The default for HEADER INITIAL or FOOTER FINAL is to print the HEADER or FOOTER, respectively.

The regular header and footer are printed at the top and bottom, respectively, of each report page. The initial header and final footer are printed only once, at the beginning and end of the report instead of the regular header and footer. At the end of the report, if there is not enough room for the final footer (which could happen if the final footer uses more lines than the regular footer), then the regular footer is printed on the last page of data and the final footer is printed on a separate page. Each header or footer contains one or more line specifications as defined below (except for NEW PAGE commands). The text of headers and footers is generated one time, at the beginning of report execution, and reprinted at the top of each page. Page numbers, if included in headers and footers, are substituted correctly. Data from the database used in headers or footers come from the first input record only.

INPUT

Syntax

INPUT input-list FILE filename [DELIMITER “string”] [sort-specification] [aggregation].
input-list ⇒ input-column [, input-column] ...
input-column ⇒ name [CHARACTER | NUMERIC] [alias]
filename ⇒ “string” [/ “string”] …

Used In

Report, Property, Calculation

The INPUT statement reads data from a file into CVIR variables. Each column name in the input-list (or alias if used) becomes a variable in the script. If the column name is a NASIS data element name, the data type for the column is the same as the elements. If not, either CHARACTER or NUMERIC must be specified.

The file name is entered in quotes. The whole file path can be within one pair of quotes, or there can be several parts within quotes and separated by a slash. A full path name is required if you are using a file that you created on your computer. If just the file name part is supplied, the file must be in the NASIS installation under the “data\input files” folder.

Examples

```
INPUT col1, col2 FILE ”lookup.data”.

INPUT areaname, areaacres FILE “C:\My Documents\datafile”.
```

The first example uses just a file name, so it is presumed to be a file distributed with NASIS. The second example uses a file in the My Documents folder.

Input from a file can be aggregated, as described above for queries, to produce single- or multiple-valued variables. If the INPUT statement precedes any queries in a script, the report has a cycle for each input record just as if a query were used. A BASE TABLE declaration cannot be used in this case. If the INPUT appears after a query, the aggregation for the INPUT is assumed to be global, similarly to a parameterized query.

The input record must be in ASCII character format. A delimiter follows each data value in the input record. Any character string can be specified as the delimiter. The default is the “pipe” character, “|”.

INTERPRET

Syntax

INTERPRET rule [, rule] … [MAX REASONS max_value] [MAX RULEDEPTH max_value] [sort-specification] [aggregation]

rule ⇒ ["site_name" :]"rule_name"

max_value ⇒ { number | variable }

Used In

Report

Generates interpretations for inclusion in a report. One or more rules can be specified, and the interpretation values are computed during each cycle of the report. The interpretations are produced for the report's BASE TABLE. The same table must be used as the base table for the properties used in the interpretation.

Note that the process for generating interpretations is not the same as in NASIS 5. Interps were formerly written to a temporary database table and retrieved with a query. In NASIS 7, the interp generator works more like a secondary query. During each report cycle, interpretation data is generated for a single row. If the base table is Component, interps are generated for one component at a time. The results become report variables as listed below. Sorting and aggregating are available on the set of interp results for each component. To aggregate above the base table, the data must be put into a TEMP Table and then you can aggregate.

If the base table is Component and you want to aggregate to the mapunit level, all the data from the interpretation is placed in a TEMP Table and then reselected in a subquery and reaggregated.

The rule list specifies rules whose values are generated. Each rule name is written as "*Site Name*": "*Rule Name*". Site Name is the NASIS site that owns the rule, and Rule Name is the name of the rule. Each name must be in quotes. This can be written as just "Rule Name" since rule names areas unique. Any rule in the NASIS database can be used in a report. If a PARAMETER is defined with the option ELEMENT *rule.rulename*, the parameter name may be used as a rule list.

The optional phrase MAX REASONS can be used to limit the number of reasons (subrules) whose results are returned from the interpretation. All subrules are used to derive the interpretation results; this only limits how many are returned in report variables. If the number of reasons is zero or this phrase is omitted, all subrule results are included, even the insignificant ones. If the number is greater than zero, the highest n significant subrule results are returned. Significant means non-zero values for limitation type interps or values other than 1 for suitability-type interps. The subrules are always sorted so the most significant values are first.

Example

```
INTERPRET "ENG - Shallow Excavations".
```

Generates the "Shallow Excavations" interpretation for each component selected by the report query and returns its results along with all subrule results.

The optional phrase MAX RULEDEPTH can be used to limit the number of levels of subrules returned. A rule depth of 0 uses only the main interp rating and no reasons. Depth 1 uses the main interp and its first level of reasons. If MAX RULEDEPTH is omitted, subrules to the maximum depth are returned.

Example

```
INTERPRET "NSSC_Pangaea": "FOR-Harvest Equipment
Operability", "NSSC_Pangaea": "FOR-Log Landing Suitability"
MAX REASONS 5.
```

Generates results for two interpretations and returns up to 5 subrule results for each rule. Only non-zero subrule results are returned. Notice that the rule names are fully qualified by NASIS site name.

Using Interpretations in Reports

The results of the INTERPRET command are placed in report variables and aggregated according to the aggregation rules for secondary queries (see sorting and aggregation). Unless MAX RULEDEPTH 0 is specified, the interp generator produces more than one value in each variable. Typically, you will need to specify the NONE aggregation type for each column you want to use in the report because the default aggregation type is UNIQUE. Crosstab aggregation is also available for reports that use more than one interpretation.

The report variables produced by the INTERPRET statement are shown in the following table.

Variable name	Description
PrimaryRuleInterpRuleID	The rule id of the top level rule.
PrimaryRuleInterpRuleName	The name of the top level rule.
InterpRuleID	Rule id of the rule or subrule that produced the rating values.
InterpRuleName	Name of the rule or subrule.
InterpRuleDepth	An indicator of the depth of the rating, where 0 is the top level.
InterpRuleResultSequence	The sort sequence of the ratings within a top level rule.
RatingValueLowLow	The fuzzy value of the minimum rating for the rule or subrule.
RatingClassNameLowLow	The rating class name of the minimum rating.
RatingValueLowRV	The fuzzy value of the minimum of the representative values of the ratings.
RatingClassNameLowRV	The rating class name of the minimum of the representative values of the ratings.
RatingValueHighRV	The fuzzy value of the maximum of the representative values of the ratings.
RatingClassNameHighRV	The rating class name of the maximum of the representative values of the ratings.
RatingValueHighHigh	The fuzzy value of the maximum rating for the rule.
RatingClassNameHighHigh	The rating class name of the maximum rating.
NullPropertyData	True/false indicator that null data was produced by a Property
DefaultPropertyData	True/false indicator that default values were used to replace null values from a Property
InconsistentPropertyData	True/false indicator that inconsistent data were detected from a Property.

Rating values can be printed either as fuzzy values (numbers between 0 and 1), as rating class names, or both. The values of the interp variables are sorted on InterpRuleResultSequence, which is a number assigned by the interpretation engine such that each subrule comes out after its parent rule, with the most significant rating values first. InterpRuleDepth can be used with the NEST option (described in column layout specifications under the SECTION statement) to print subrules indented below their parent rules.

MARGIN

Syntax

MARGIN [LEFT *number* [IN]] [RIGHT *number* [IN]] [TOP *number* [IN]] [BOTTOM *number* [IN]] .

Used In

Report (text style only)

Example

```
MARGIN TOP 1 inch BOTTOM 1 inch.
```

Defines margins for the pages of text-style reports. Defaults are one-half inch for all margins. If margins are specified with IN, INCH, or INCHES, they are measured in inches; otherwise, they are in lines or characters. The relationship between lines, characters, and inches is defined by the PITCH specification.

The margin specifications are used to determine how much text can be placed on a page of output. However, the text file produced by NASIS does not have blank lines for top and bottom margins, nor does it have blanks spaces for the left and right margins. The user is responsible for setting appropriate margins in the application used to display the report.

If the page length is UNLIMITED, the top and bottom margins are ignored. If the page width is UNLIMITED, the left and right margins are ignored.

PAGE

Syntax

```
PAGE [ LENGTH { number [IN] | UNLIMITED } ]
[ WIDTH { number [IN] | UNLIMITED } ] .
PAGE PAD
      line-specification ...
END PAGE PAD .
```

Used In

Report (text style only)

Example

```
PAGE WIDTH 144 LENGTH 88.
PAGE PAD
     USING normal_template.
END PAGE PAD.
PAGE WIDTH UNLIMITED LENGTH UNLIMITED.
```

The Page Length/Width statement defines the size of the assumed page for report output. The default size is length 11 inches and width 8.5 inches. If sizes are specified with IN, INCH, or INCHES, they are measured in inches, otherwise they are in lines or characters. The relationship between lines, characters and inches is defined by the PITCH specification.

NASIS does not control the final appearance of the report because it is displayed in an application of the user's choosing. The user may have to set the page size, margin, and font in that application in order to produce printed output that matches the desired pagination.

Length can be specified as UNLIMITED, which means that the whole report is treated as a single page. This is used in reports that are intended for onscreen display or for saving as text; if the output is sent to a printer, however, there will not be headings on each page.

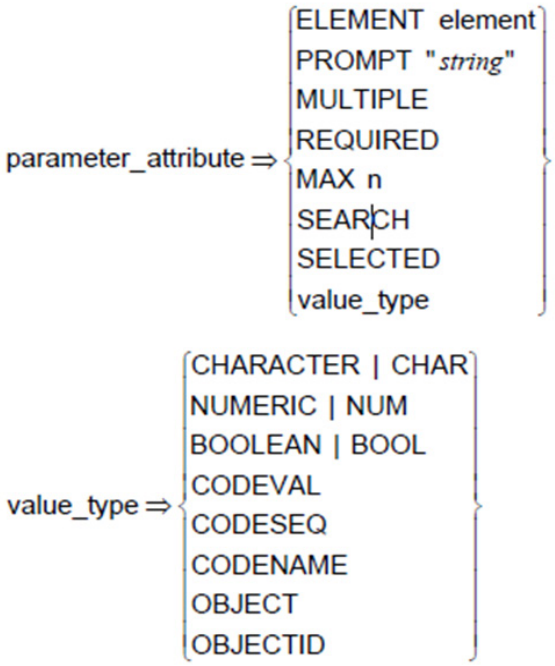
Width can also be specified as UNLIMITED, which means that report lines are as long as the data requires. This is useful when the report output is saved as text, but sending the output to a printer is typically not be desirable.

PAGE PAD is used when lines are used to fill any unused space at the end of a page or when the FILL command is used. The line specification in the PAGE PAD block is printed instead of the default padding, which is a blank line. If the block contains more than one line, the whole block of lines is printed repeatedly to fill the required space.

PARAMETER

Syntax

PARAMETER name [parameter_attribute] ... .



Used In

Report

Example

```
PARAMETER asym ELEMENT areasymbol PROMPT "Soil survey area
symbol".
```

A PARAMETER allows the user to customize the report script through a dialog. Parameter names are variables in the report script, such as variables created by the DEFINE statement, and may have zero or more values. Parameters are commonly used in WHERE clauses, to provide a rule name in the INTERPRET statement, or for column names in a CROSSTAB.

The PARAMETER definition statement is normally placed at the beginning of the report script. For compatibility with older report scripts, the statement may begin with a # symbol as was previously used for comments. The # symbol, however, is no longer required. Following the parameter name, one or more attributes may be specified to customize the parameter dialog. The attributes are:

ELEMENT means that the parameter takes the same values as the named data element. If the element has a choice list, that list appears in the parameter dialog. The element's data type also applies to the parameter value(s). The element name should be written as *tbl_nm.elm_nm* to be sure that the name is unique.

PROMPT provides a label for the input field in the parameter dialog. This can give the user hints on how to fill in the parameter value. If no prompt is provided, the Label field from the ELEMENT definition is used as the prompt. If neither PROMPT nor ELEMENT is provided, the parameter name is used as the prompt.

MULTIPLE means that more than one value can be entered for the parameter. If a choice list is used, multiple choices can be selected. The result of the parameter dialog is a multiple-valued variable.

REQUIRED means that at least one value must be entered for the parameter. This option is useful if the parameter is intended to be used in a query to select data and you want to ensure that something is selected.

MAX n implies MULTIPLE and also puts an upper limit on the number of values that can be selected.

SEARCH means that a choice list is built for use in the parameter dialog by searching the database for all unique values entered for the specified data element. The ELEMENT attribute must be used with SEARCH. Note that it can take some time to build a choice list if the element is in a table with many rows.

SELECTED is like SEARCH but only searches the data in the current selected set. This presents the user with a list of choices that could actually appear in the report. An example is a choice list for crop name. Normally, this lists all crop names in the domain, but the choice list if SELECTED is used includes only crops that actually occur in the selected set.

The value type option allows the type of the parameter to be specified when the ELEMENT attribute is not used or when code conversions are needed. The type tells the parameter dialog how to format the parameter value. Only one value type may be specified. The allowed types are:

CHARACTER means that the value is composed of any printable characters. This is the default if neither type nor ELEMENT is specified.

NUMERIC means that the user must enter a number.

BOOLEAN means that the parameter dialog displays a toggle button instead of a data entry field. The parameter's value is numeric. It contains a 1 or zero indicating whether or not the user toggled the button.

CODEVAL can be used when the parameter refers to a data element that uses codes. This option specifies that the parameter value is returned as the code's value, although the choice list contains code names. The default for coded elements is CODEVAL.

CODESEQ can be used when the parameter refers to a data element that uses codes. The parameter value is returned as the code's sequence number.

CODENAME can be used when the parameter refers to a data element that uses codes. The parameter value is returned as the code's name.

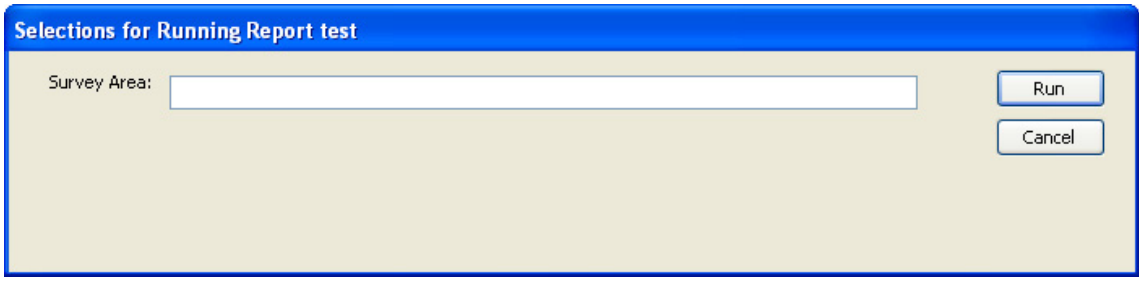
OBJECT means that the parameter is an object name, such as a rule or property name. The parameter dialog displays a choice list for selecting NASIS site and object names. The parameter must refer to an element in the root table of a NASIS object, typically the name column. The value returned is in the format used in the DERIVE and INTERPRET statements, namely *"site":"name"*. If OBJECT is used with the MULTIPLE option, the user can select multiple object names.

OBJECTID produces the same kind of choice list as OBJECT, but the value returned is the record ID of the selected item(s). This option is used when you want to query for a specific item or items, as opposed to the OBJECT option which returns a value that is not usable in a query WHERE clause.

The following fragments of report scripts illustrate the use of parameters:

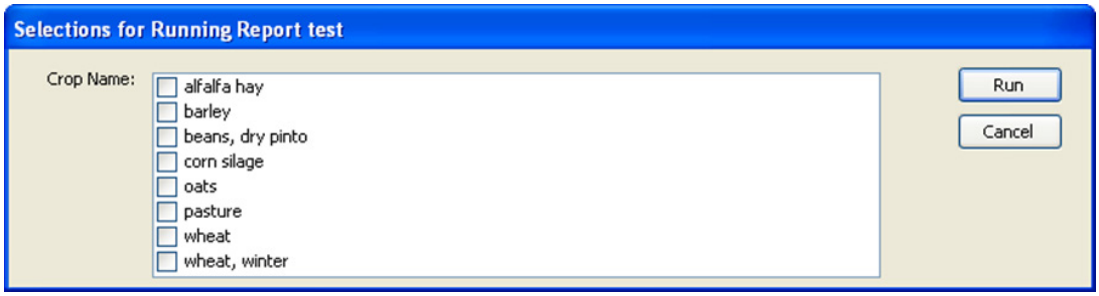
```
PARAMETER aname  ELEMENT area.areaname  PROMPT "Survey
Area".

EXEC SQL select ... WHERE area.areaname = aname and ...
```

This asks the user to provide a survey area name, which is then used in a query to get records for the selected area. The parameter dialog would look like:



```
PARAMETER crops ELEMENT dmucropyld.cropname MULTIPLE SELECTED.
…
…CROSSTAB BY dmucropyld.cropname VALUES crops
```

This example allows the user to select one or more crop names from a choice list based on the contents of the selected set. The names are used as column headings in a crop yield report. The prompt is the label for the element *dmucropyld.cropname*, which is "Crop Name", as shown:



PITCH

Syntax

PITCH [HORIZONTAL number] [VERTICAL number].

Used In

Report (text style only)

Example

```
PITCH HORIZONTAL 17 VERTICAL 8.
```

Defines the character spacing in characters per inch or lines per inch. The default is horizontal 10 characters per inch and vertical 6 lines per inch, which correspond to a 12-point fixed-width font, such as Courier. The pitch specifications are used to determine how much text will fit on a page of output. They are used in combination with the width, length, and margins for a page as specified in the PAGE statement.

SECTION

Syntax

SECTION [section-name] [keep-option] [condition] [HEADING output-specification ...] [DATA output-specification ...]
END SECTION .
section-name ⇒ name

Used In

Report

Example

```
SECTION WHEN LAST OF musym KEEP WITH main DATA
AT 40 "----------".
AT 40 total_acres width 10 decimal 2.
END SECTION.
```

A report section defines a block of report output that is produced as a unit. A section can be unconditional, meaning that the section's data block is printed on each cycle of the report's main query, or it can be printed only when certain conditions occur. A report can have any number of sections. The sections are printed in the order determined by their conditions, as discussed below under Section Conditions.

For a simple example, imagine a report script having a section "A" that prints the map unit symbol and map unit name followed by a section "B" that prints the component name. Section B is unconditional, and section A prints whenever the value of the variable "musym" changes. This would be defined in the following manner:

```
SECTION A WHEN FIRST OF musym
DATA
AT LEFT musym, muname.
END SECTION.

SECTION B
DATA
AT LEFT compname.
END SECTION.
```

The output of the report might be:

```
12A Hamerly-Vallers complex, 0 to 2 percent slopes
Hamerly
Vallers
Hamre 26A Windsor loamy sand, 0 to 3 percent slopes Windsor
```

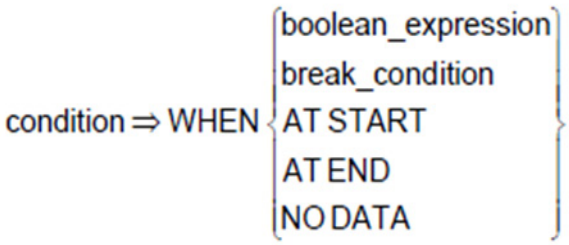
This would be produced from a query returning 4 records for the two map units. The first map unit has three components, and the second map unit has one component. Because section A has a "first of" condition, it is printed before unconditional sections when a new value of musym is encountered. Then section B is printed for each input record until a change in musym occurs. Then section A is printed again, and finally section B is printed for the last record.

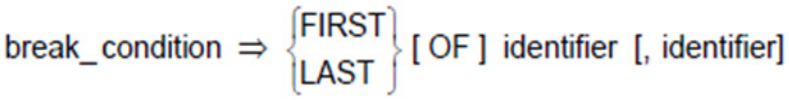
To define a section, specify one or more of the following features, each of which is discussed in more detail later. Note that XML-output style has no concept of a page, so page layout features, such as KEEP and HEADING, are ignored.

1. A section can be given a name. Names are used in the KEEP option and can be useful as documentation.
2. KEEP controls the splitting of the section when the end of a page is reached.
3. A condition specifies when the section is used. If no condition is provided, the section appears for each report cycle.
4. If a HEADING block is provided, it prints at the top of the report page after the general header. If the section has no condition, the heading prints on every page; if the section has a condition, the heading only prints if the condition is true when it is time to start a page. The heading block contains one or more output specifications. If any data element values are printed in a heading, they come from the record being processed at the time the heading prints. (Note that this differs from the use of data in headers and footers).
5. If a DATA block is provided, it prints on each report cycle for which the condition holds. The data block contains one or more output specifications, and each of them can have an IF condition attached. The actual number of lines of output produced by a section is determined by many factors, including the conditions, the number of values stored in the variables being printed, and the length of text fields.

SECTION: Conditions

Syntax





Example

```
SECTION WHEN type == 2
```

A condition can be an ordinary Boolean expression based on data from the database or report variables. In this case, the section prints whenever the condition evaluates to True. Boolean expressions are described under the DEFINE statement.

Another form of the condition detects control breaks in the report data. This type of condition begins with the keyword FIRST or LAST. At least one of the identifiers in the break condition should be a data element in the sort key for the main report query. A control break occurs when the value of any specified element, or of any element higher in the sort key, changes. The choice of FIRST or LAST in the break condition determines which data are used for the lines printed in the section. With FIRST, the first record with the new value of the control variable is used. LAST uses the last record with the old value. The LAST condition would be used for printing subtotals for a group of records. FIRST would be used for printing a heading line before a group of records.

The remaining conditions are used for special conditions that occur no more than once in a report.

The AT START condition means that the section prints before any other sections (but after the headers); and an AT END section prints after the last data record (but before the footers). The default for these sections is no printing.

A NO DATA section prints only if there are no input records and could be used print a message such as “No data found”. If the NO DATA section is not used and there is no input, no report output is produced. Instead, a warning dialog is displayed to the user.

When a heading block is specified in an unconditional section, the result is easy to visualize: The heading lines print on each report page following the page header. The headings appear in the order that the sections are defined. To reduce confusion, it is a good idea to include all unconditional headings in a single section that is first in the script. In a simple report, both headings and data can be specified in the same unconditional section.

The operation of headings in conditional sections can produce expected results. These headings print only if a page break occurs while the conditional section is being printed. It helps to arrange for a page break to occur just before printing the conditional section. This feature can require some trial and error to get the desired results.

Heading lines can contain references to data elements or variables, whose values print in the heading. Note that headings are generated each time a new page begins, so the heading contains the values in effect at the time it prints. In particular, a LAST OF section uses values from the last record before the control break and a FIRST OF section uses values from the new record (the one causing the control break). Note, however, if a LAST OF section (or any other type of section) causes a page break, all the headings on the new page will use data from the new record.

The order of processing for section conditions is:

1. AT START (only once per report)
2. FIRST OF (per report cycle)
3. Other sections, in the order they appear in the script
4. LAST OF (per report cycle)
5. AT END (only once per report)

SECTION: KEEP option

Syntax

keep-option ⇒ { NO KEEP | KEEP WITH section-name [, section_name] ... }

Example

```
SECTION B KEEP WITH A
```

The KEEP option controls what happens when the end of a page is reached while a section is being printed. Without a KEEP option, the default behavior is to allow a page break to occur after printing all the lines defined for one section occurrence. If the DATA block contains more than one line specification, or if continuation lines are needed for long text fields, these output lines are kept together on a page. The NO KEEP overrides this by allowing page breaks between lines of a section, although text continuation is kept on a page if possible.

The KEEP WITH option specifies other sections with which this section is linked. This means that when a section directly follows an occurrence of one of its "keep with" sections, the data block for the new section occurrence must fit on the same page as the last line of data in the "keep with" section. If there is not room, a page break is inserted before the last keep block of the named section.

KEEP options are ignored in HTML-style output. All page layout is controlled by the style sheet applied to the output of the report generator.

SECTION: Output Specifications

Syntax

output-specification ⇒ [IF expression] line_content

line_content ⇒ { SKIP *number* {LINES | INCHES}.
FILL *number* {LINES | INCHES}.
NEW PAGE.
INCLUDE subreport [(argument [, argument] ...)]
USING template_name column_spec [, column_spec] ....
at_statement
element_statement }

subreport ⇒ ["*site_name*" :]"*report_name*"

argument ⇒ { variable
element
literal }

Examples

```
SKIP 2 LINES.

AT LEFT musym WIDTH 8, muname WIDTH 50.

IF comp_pct > 10 USING comp_tmpl compname, slope_l,
slope_h.

ELEMENT "tr" musym TAG "td", muname TAG "td".

INCLUDE "MLRA10_Office":"Flood Subreport" (dmudbsidref,
coiid).
```

An output specification is used either to control spacing on the page or to produce actual report output. Output specifications can be either conditional or unconditional. When the IF clause is used, the IF expression is evaluated each time the section is processed. The expression follows the same rules as expressions for the DEFINE statement. If it results in a True (non-zero) value, the output content is produced. If the value of the expression is False (a null, a zero, or an empty character string) nothing is output. Without the IF clause, the output is always produced when its section is printed.

The output content is sometimes called a "logical line" because it is a single unit of output, even though it may include several "physical" lines on the report page. For example, a logical line containing a text field may require several lines on the page to print all the text. Depending on the KEEP rules, a whole logical line is normally kept together on one page unless the text requires more than a full page to print.

The line_content portion of the command describes the output.

1. The SKIP command produces the specified amount of blank space. Either LINES or INCHES must be specified for the amount to be skipped. When page formatting is in effect, skip lines are not carried over past the bottom of a page. SKIP has no effect in XML-style output.
2. The FILL command is like the SKIP command, but it fills the specified space with repetitions of the PAGE PAD block.
3. NEW PAGE fills out the page with repetitions of the PAGE PAD, then prints the footer, starts a new page, and prints the header. If NEW PAGE occurs at the very end of a report, the report generator ignores it and does not print an extra blank page. NEW PAGE has no effect in XML-style output.
4. The INCLUDE command runs another report and inserts its output as a logical line in the first report. Parameters may be passed to the subreport, and they must correspond with variables in the subreport's ACCEPT statement. Typically, a record key would be passed as a parameter, which would be used by the subreport to query for information related to that record. See Using Subreports for more detail.
5. The USING statement specifies a template to serve as a format for the output. The column specifications in the USING statement are matched to the FIELD keywords in the template. The element or variable specified in the column spec is printed with the formatting defined in the template. But any formatting options specified in USING override the corresponding options in the template. If USING does not have as many columns as there are FIELDs in the template, the remaining fields are printed as blank. USING may not have more columns than the template has FIELDs. Columns in the USING statement may not use the ARRAY or FIELD options.
6. A text-style output line is created with the AT statement as described below.
7. HTML-style output is created with the ELEMENT statement.

Line Specifications

This option controls each line of data with key terms and IF THEN conditions. The most common specifications are:

- USING (template name)
- SKIP line.—controls spacing
- NEW PAGE.—creates a new page

- INCLUDE (report name).—adds data from a subreport
- AT Statement.—controls the position on the page.

Examples

```
SKIP 2 LINES.

AT LEFT musym WIDTH 8, muname WIDTH 50.

IF comp_pct > 10 USING comp_tmpl compname, slope_l,
slope_h. INCLUDE "MLRA10_Office":"Flood Subreport"
(dmudbsidref, coiid).

AT CENTER title WIDTH 20 CENTERED; AT RIGHT date WIDTH 12.

COLUMN SPECIFICATIONS
```

If the condition is True, the output content is produced. If the value of the expression is False, nothing is output. Where the IF clause is not used, the output is always produced when its section is printed.

This option identifies exactly what is printed at a particular spot in a report. These specifications are added after each column of data. Some of the most common specifications are WIDTH, DECIMAL, ALIGN, SEPARATOR, NO COMMA and SUPRESS DUPLICATES.

Example

```
AT LEFT musym width unlimited SEPARATOR "|", muname width
unlimited.
```

SEPARATOR is used to separate columns. SEPARTOR precedes the column of stat; therefore, if you do not want a SEPARTOR at the beginning of the table, you must specify "NO SEPARATOR" as a column specification and if you want a vertical line at the end of the table you have to create a null field "" with SEPARTOR.

AT Statement

Syntax

at_statement ⇒ AT position [alignment] column_spec [, column_spec] …
[; AT position [alignment] column_spec [, column_spec] …] … .
position ⇒ { number [IN] | LEFT | RIGHT | CENTER }
alignment ⇒ { TOP | BOTTOM | SAME }

Used In

Reports (Text Only)

Examples

```
AT LEFT musym WIDTH 8, muname WIDTH 50.

AT CENTER title WIDTH 20 CENTERED; AT RIGHT date WIDTH 12.
```

The AT statement specifies one or more groups of columns to be placed at specific positions in an output line. An AT position can be a number, expressed in characters or inches from the left margin; or it can be the left, right, or center of the line, relative to the margins. The position may be followed by an alignment option, which defines where this group of columns appears vertically on the page relative to the previous AT group. See the examples below. If no alignment is specified, the default is TOP.

Following the position and alignment, one or more columns are specified. These columns print adjacent to each other in order from left to right. They each occupy the number of characters specified by its WIDTH. A column that has WIDTH UNLIMITED uses as many characters as needed to output the data, which may vary from line to line.

If the columns are not supposed to be adjacent, a new AT keyword and position may be used. The list of columns following the AT begin at the new position. A semicolon must separate AT groups as shown in the syntax. Note that when more than one column spec follows an AT RIGHT or AT CENTER, the group of columns is first strung together and then is right justified or centered as a unit.

Another application for an AT group occurs when printing data from array variables. Within an AT group that contains array variables, corresponding values in each array variable always come out on the same line. If a value in an array is text that wraps around to a new line, blanks are inserted in the other columns as needed to maintain alignment across columns. There is no alignment across columns for columns in different AT groups. Text wrapping can cause data from different array positions to appear on the same line (which is desirable in some reports). The following example illustrates this:

This line specification uses columns “name”, “age”, and “score” from a query with aggregation type NONE, so each column contains an array of values. The column “text” comes from a different query and has only one value, which is a long text string.

```
AT LEFT name width 12, age width 6, score width 7; AT 28 text
width 30.
```

This might produce the following output. Because it is in a different AT group, “text” wraps across several lines, and is not associated with any one of the name lines. Where a name wraps, however, the associated data stays in alignment.

```
Jones           30    5.9  This group of people has
Abercrombie-    52    5.4  responded to all the
Fitch                      surveys conducted since
Smith           27    6.1  1983.
Martinez        41    5.7
```

The line with the name “Abercrombie-Fitch” requires two lines of output because the name doesn’t fit in 12 characters. The age and score are printed on the first of these lines, and a blank appears beneath them due to wrapping in the first column.

In some cases, this might not be the desired output. If you want the age and score to appear lined up with the end of the name rather than the beginning, use the alignment option. Age and score would need to be in a separate AT group using BOTTOM alignment, meaning that they line up with the bottom of the previous AT group. The following example shows this:

```
AT LEFT name width 12; AT 13 BOTTOM age width 6, score width 7;
AT 28 text width 30.
```

This version would produce the following output.

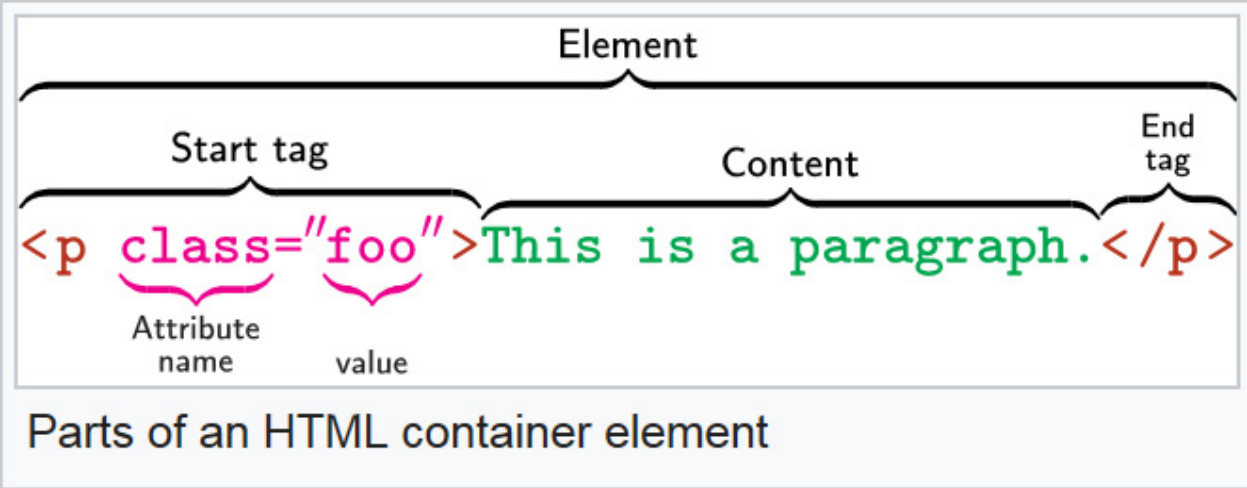
```
Jones          30    5.9   This group of people has responded to
Abercrombie-                all the surveys conducted since May,
Fitch          52    5.4
Smith          27    6.1   1983.
Martinez       41    5.7
```

There is a third possible alignment option, SAME. This is used in cases where there are three or more AT groups, the second group has BOTTOM alignment, and both first two groups could have wrapping of long text. Then there are three possible places for the third AT group to line up: the original top line of the first group, the bottom of the first group (which is the same as the second group), or the bottom of the second group. These three positions correspond to the alignments TOP, SAME, and BOTTOM. For an example in NASIS, see the national manuscript report Table E2.

ELEMENT Statement

Syntax

output-specification ⇒ ELEMENT [OPEN | CLOSE] element-name
[attribute [attribute] …] [value-tag] [column-spec [, column-spec] …]
element-name ⇒ “string”
attribute ⇒ (“string”, { “string” | variable })
value-tag ⇒ VALUETAG “string” [attribute [attribute] …]



Parts of an HTML container element

Examples

```
ELEMENT “para” ATTRIB (“role”, “subhead”) musym, “: “,
muname.

ELEMENT “tr” musym TAG “td”, muname TAG “td” ATTRIB (“role”
“namecol”).
```

The ELEMENT statement creates output in Extensible Markup Language (XML) format. XML is an industry standard for exchanging information on the web. An introduction to XML is available from W3C, the web-standards organization. The element is an individual component of an HTML document or webpage once it has been parsed into the Document Object Model.

An HTML document is built up out of elements and attributes. The elements are all nested with the HTML element as the outer level, or root element. Each element may also contain certain attributes. A <P> element might contain the text node "Hello, World!" and also a style attribute. Attributes are in key and value pairs, so the style attribute would be set to some value.

HTML, the standard language for webpages, is a subset of XML and can be produced with the ELEMENT statement. The complete specification for HTML 5.1, 2nd edition is online at the following link.

http://www.w3.org/TR/html51/

Many references have been written about HTML. An example of a simple online reference is available at the following link.

http://www.simplehtmlguide.com/

When HTML is specified as the output format for a report, NASIS converts the XML generated by ELEMENT statements into HTML so that the output can be viewed directly in a browser. The conversion works best if the report follows the conventions documented in Appendix 1, which are based on a documentation standard called DocBook.

The ELEMENT statement includes an element name that appears in the opening and closing tags that surround the content of the element. The first example above uses the element name "para", which is the DocBook tag for a paragraph, and a class attribute of "subhead". The content of the element is three items: a value of musym, a colon-space, and a value of muname. The output generated by this element might look like this:

```
<para role="subhead">12: Alpha silt loam, 5 to 8 percent
slopes</para>
```

Any attributes applicable to an element can be added with the ATTRIB option. Within the parentheses, you provide an attribute name and the value. In this example, the attribute "role" is a standard DocBook attribute that links to a style sheet where the formatting for elements of role "subhead" is defined. An element statement may contain several attributes.

Because XML describes structure as well as data, there is commonly a need to produce elements within elements. The CVIR language provides some ways to do this. The first way is the TAG attribute, illustrated in the second example above. When TAG is used in a column-spec, XML tags are produced around that column's data. Following TAG, additional ATTRIB options can be specified. Those attributes go in the column's tag rather than the outer element tag. The output might look like this:

```
<tr><td>12</td><td role="namecol">Alpha silt loam, 5 to 8
percent slopes</td></tr>
```

This output is a typical DocBook specification for a row of a table (abbreviated "tr") containing two table data ("td") columns. A more common use would be to put the element definition in a TEMPLATE statement, then specify the variables to be output with a USING statement, as in the following example, which produces the same output as the example above.

```
TEMPLATE row1 ELEMENT "tr" FIELD TAG "td", FIELD TAG "td"
ATTRIB ("role" "namecol").
USING row1 musym, muname.
```

A further level of XML structure can be applied when the variables being printed in the ELEMENT statement are arrays. The VALUETAG option is similar to the TAG option, except that it places a tag around each value of the array. TAG would place the tag around the whole set of values. You can use both TAG and VALUETAG to get a nested tag effect. The following example uses the *hzname* variable, which has multiple values:

```
TEMPLATE row2 ELEMENT "tr" FIELD TAG "td", FIELD TAG "td"

VALUETAG "para" ATTRIB ("role" "namecol").

USING row2 compname, hzname.
```

This example puts each horizon name within a "para" tag so that it appears on a separate line. The example uses the "namecol" role to get the right formatting for that column. The output would be like the following (indentation has been added to make it more readable).

```
<tr>
   <td>Alpha</td>
   <td><para role="namecol">A</para>
      <para role="namecol">B</para>
      <para role="namecol">C</para>
   </td>
</tr>
```

The following example has the VALUETAG nested within the TAG option. It is also possible to nest TAG options within a VALUETAG. In such cases, the outer tag is repeated for each set of inner tags, as in the following example.

```
TEMPLATE row3 ELEMENT "table" VALUETAG "tr" FIELD TAG "td",
FIELD TAG "td" ATTRIB ("role", "number").
USING row3 hzname, hzdept_r.
```

This example produces a complete table that displays a set of horizon names and depths, which are assumed to be aggregated into multiple-valued variables. Each pair of values for *hzname* and *hzdept_r* becomes a table row enclosed in a "tr" tag. The output might be the following.

```
<table>
   <tr>
     <td>A</td>
     <td class="number">0</td>
   </tr>
```

```
        <tr>
          <td>B</td>
          <td class=”number”>15</td>
        </tr>
        <tr>
          <td>C</td>
          <td class=”number”>45</td>
        </tr>
      </table>
```

This type of output works best if all the variables have the same number of values. If they do not, then the rows don't all have the same number of columns, which produces a poor looking page when output to HTML.

The examples so far have shown how to produce XML tags nested up the three levels deep, using ELEMENT, TAG, and VALUETAG. Commonly, it is necessary to add even more levels for larger structures. The last example showed a very simple table, but often a table has too much information to fit in one ELEMENT statement. Also, a <table> element would be inside the main `<section>` element of a typical DocBook document. These larger structures typically encompass most, if not all, the data in a report; so, the opening and closing tags cannot be produced in a single statement.

For more complex structures, we use conditional sections together with the ELEMENT OPEN and ELEMENT CLOSE forms of the statement. ELEMENT OPEN produces only the opening tag, and it might be placed in a section with a condition of WHEN AT START or WHEN FIRST OF. The ELEMENT CLOSE statement produces the corresponding closing tag. It would be in a section with condition WHEN LAST OF or WHEN AT END. Following is an example of a part of a script to produce a DocBook table:

```
SECTION WHEN AT START
   DATA
     ELEMENT OPEN "section" ATTRIB ("label", "SoilReport").
     ELEMENT "title" reporttitle.
     ELEMENT OPEN "table.
     ELEMENT OPEN "thead.

      elements for table header …

     ELEMENT CLOSE "thead.
     ELEMENT OPEN "tbody.
END SECTION.

SECTION
      DATA
           USING row1 compname, hzname, etc.
```

```
      END SECTION.

SECTION WHEN AT END
  DATA
    ELEMENT CLOSE "tbody".
    ELEMENT CLOSE "table".
    ELEMENT CLOSE "section".
END SECTION.
```

Each ELEMENT OPEN must have a matching ELEMENT CLOSE.

There are no rules about when to use attributes and when to use child elements. Attributes are handy in HTML, but you should try to avoid using them in XML; otherwise, use child elements.

Tags are used to mark up the start and end of an HTML element.

An attribute defines a property for an element, consists of an attribute/value pair, and appears within the element's start tag.

Some of the problems with attributes are:

- Attributes cannot contain multiple values (child elements can),
- Attributes are not easily expandable (for future changes),
- Attributes cannot describe structures (child elements can),
- Attributes are more difficult to manipulate by program code, and
- Attribute values are not easy to test against a Document Type Definition. (Further information regarding DTD is available online. An example of a site that describes DTD is at the following link. https://www.w3schools.com/xml/xml_dtd_intro.asp.

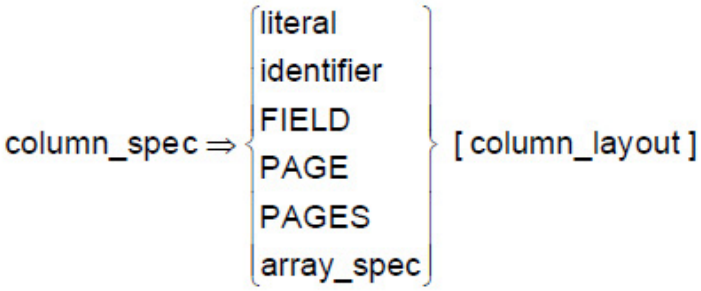
If you use attributes as containers for data, you end up with documents that are difficult to read and maintain. Try to use elements to describe data. Use attributes only to provide information that is not relevant to the data.

One exception: Sometimes it is effective to assign ID references to elements. These ID references can be used to access XML elements in much the same way as the NAME or ID attributes in HTML.

NOTE: Metadata (data about data) should be stored as attributes, and that data itself should be stored as elements.

Column Specifications

Syntax



array-spec ⇒ ARRAY (column-spec [, column-spec] ...)

The column specification identifies exactly what is printed at a spot in a report. A column can print data from a literal, variable, data element, or page number. It can also be a compound column (ARRAY) consisting of one or more subcolumns. In a template definition, the keyword FIELD is used as a place holder, and the actual element, variable, or literal is supplied later.

If a variable or element is printed, its value at each report cycle prints according to the layout options. If a literal is used, it prints the same value each time. The keywords PAGE and PAGES generate page numbering and are normally used in headers or footers. Wherever the word PAGE occurs, the number of the current page is substituted before column layout options are applied. The keyword PAGES is replaced by the total number of pages in the report, as in "Page n of m".

When the ARRAY specification is used, a group of one or more columns is printed repetitively. The columns are printed using the same format. Array columns are used only with crosstab reports. The number of columns printed equals the number of column values in the crosstab, times the number of column specs in the ARRAY spec. The printing sequence is to print all the columns listed in the ARRAY spec, then repeat for the number of crosstab values. Any column layout options listed outside the parentheses of an ARRAY spec apply to all columns within the parentheses unless overridden by layout options that are inside the parentheses and apply to an individual column.

In the description of the CROSSTAB Statement, there is an example that produced the data shown in the following table. The variables *areaname* and *acres* are arrays with 3 values each.

musym	muname	areaname			acres		
10A	Alpha loam, 0 to 3	X	Y	Z	100	300	400
10A	Alpha loam, 0 to 3	X	Y	Z	200		500
10B	Alpha loam, 3 to 6	X	Y	Z	600	700	
10B	Alpha loam, 3 to 6	X	Y	Z		800	

The following example shows one way these data could be printed. Ignoring column formatting details for the moment, these line specifications for heading and data would produce the report fragment shown.

```
HEADING
     AT 1 musym LABEL, muname LABEL, ARRAY(areaname).
DATA
     AT 1 musym, muname, ARRAY(acres, "acres").
```

musym	muname	X		Y		Z	
10A	Alpha loam, 0 to 3	100	acres	300	acres	400	acres
10A	Alpha loam, 0 to 3	200	acres	acres		500	acres
10B	Alpha loam, 3 to 6	600	acres	700	acres	acres	
10B	Alpha loam, 3 to 6	acres		800	acres	acres	

The heading line prints the labels for the data elements musym and muname, which we assume are just the column names, then the values for *areaname,* which define the groupings.

The data line prints *musym* and *muname,* this time as normal report columns, then acres and the literal "acres" as an array. The values from *acres* are paired with the word "acres" and printed in three columns. In this example, the crosstab was not set up with aggregation, so there are several blank spaces, but the literal prints anyway. The report could be made to look better by changing the crosstab or by moving the word "acres" into the heading.

When a multiple-valued variable is printed in a column that does not have an array spec, the values are printed one beneath the other in the column. It results in a set of parallel report columns for each query column, as illustrated earlier.

Column Layout Specifications

Syntax

column-layout ⇒ [WIDTH *number* [IN]]
[WIDTH UNLIMITED]
[LABEL]
[DIGITS *number*]
[DECIMAL *number*]
[SIGDIG *number*]
[ALIGN { LEFT | CENTER | RIGHT }]
[PAD "*character*"]
[INDENT *number* [IN]]
[NEST *number* [IN] PER identifier]
[NO COMMA]
[TRUNCATE]
[REPEAT]
[SEPARATOR "*string*"]
[REPLACE NULL [WITH] literal]
[REPLACE ZERO [WITH] literal]
[SUPPRESS [DUPLICATES] [BY identifier]]
[QUOTE[D] [quote-string] [ESCAPE escape-string]]
[TAG "*name*" [attribute] …
[VALUETAG "*name*" [attribute] …

attribute ⇒ ATTRIB ("string", { "string" | variable })

In HTML formatting, COL can be used to vary the size of each column dynamically.

Example

```
ELEMENT 'col' ATTRIB("width', "3*")
```

Each column in a report can use zero or more of the above layout. Each option can be used only once per column. The options are generally the same for headings and data, although some are not useful in headings. The options can be written in any order.

1. The WIDTH option overrides the default width for the data in the column. The default width is taken from the template in a USING statement, from the data dictionary for data elements, or from the string length for a literal without the REPEAT option. In text-style output, there is no default width for a variable. In XML output, the default is WIDTH UNLIMITED. If the width is followed by IN (or INCH or INCHES), the width is measured in inches as determined by the horizontal pitch. The default is to measure width in characters.

2. The WIDTH UNLIMITED option formats the output without fixed column widths. This overrides the normal word wrap function and the TRUNCATE, ALIGN, INDENT, NEST and REPEAT formatting options. The data for the column is printed in the minimum space needed to contain the entire value, preceded by the optional SEPARATOR string. Numbers are formatted with decimal places defined in the usual manner with no leading spaces or zeros. This is useful with the PAGE WIDTH UNLIMITED option for producing text-style exports and is the default for XML-style output.
3. When LABEL is specified, the value printed is not the data but rather the *Column Label* from the data dictionary for the specified element. This could be used in column headings. If LABEL is used with a literal or variable, the result is a blank.
4. The DIGITS option is used with numeric data to specify the number of digits to be printed to the left of the decimal point. The default number of digits for data elements is taken from the data dictionary. This specification is overridden if the WIDTH is given explicitly. Numeric values over 999 are printed by default with commas between groups of 3 digits. The commas are not counted as digits but do count in the column width.
5. The DECIMAL option is used with numeric data to specify the number of digits to be printed to the right of the decimal point. The default number of decimal places is taken from the data dictionary if a data element is being printed; otherwise, the default is zero. If the number of decimal digits is zero, the decimal point is not printed.
6. The SIGDIG option is used with numeric data to specify the number of significant digits in the value to be printed. The value is rounded off so that only the significant digits are shown, and zeros are added as necessary to fill out the remaining places required by the DIGITS and DECIMAL specifications. The number of significant digits specified must be greater than zero. If SIGDIG is not specified, all digits are considered significant. The following examples show the relationship of the DECIMAL and SIGDIG specifications:

Original Value	DECIMAL	SIGDIG	Result
527.36	2	3	527.00
0.456	2	1	.50
1384.2	0	2	1400

7. The ALIGN option positions the data within the column. The default is based on the data dictionary definition for elements. For variables and literals, the defaults are left alignment for character data and right for numeric.
8. The PAD option provides a character to fill out blank space in the column when the data is shorter than the column width. Padding occurs on the right if the column is aligned left, on the left if the column is aligned right, and on both sides if the column is aligned center. If the text in a column is word wrapped, padding is only applied on the last line of the text. The default pad character is a space.

9. The INDENT option positions the data a specified number of characters or inches from its alignment position. A positive indent applies to the first line of a word-wrapped string. A negative indent applies to lines after the first. In other words, for typical left aligned data, a positive indent produces first line indentation and a negative indent produces "hanging" indentation. For right aligned data, it works the same way but relative to the right edge of the column.
10. The NEST option is provided for printing interpretations. These are traditionally printed in a "nested" or "outline" format with results of each subrule indented below its parent rule. The amount of indentation increases with each level of subrule. So a nested format is really a variable indentation with the amount of indent proportional to the depth of nesting. The output of the INTERPRET command includes a column InterpRuleDepth for just this purpose. The NEST format option allows you specify an indentation of n spaces (or inches) per depth level. This can be combined with normal indentation, such as a negative indent amount for hanging indents. For example, to print an interpretation result RatingClassNameHighRV with hanging indent of 1 space for word wrapping plus nesting of 2 spaces per level, use:

```
RatingClassNameHighRV INDENT -1 NEST 2 PER InterpRuleDepth
```

11. The NO COMMA option suppresses the placement of commas in numbers larger than three digits. This is used for printing a numeric value that should not have commas, such as a year or an ID number. It is also used to avoid inappropriate commas when exporting data in a comma delimited format.
12. The TRUNCATE option determines what happens when character type data is too long to fit in a column. The default is to split the data across multiple lines with word wrapping. If TRUNCATE is specified, the data is printed on a single line and truncated to the fit the column width. Numeric type data is never wrapped; if it is too long to fit the column, asterisks are printed.
13. The REPEAT option means that the column's value is repeated as often as necessary to fill the column width. This is typically used with a literal, such as "-". The REPEAT option in this case would fill the column with dashes. The column width must be specified explicitly when REPEAT is used.
14. The SEPARATOR string prints prior to the data for the column. It is placed at the column's specified starting position. The actual data for the column starts after the separator. If this option is not included, no separator is printed. There is no option to specify a separator to the right of a field. To print a right border on a line, add a field of width zero at the end of the line and add the desired border character as its separator.
15. The REPLACE options allow the printing of some other value when a zero or null is found. This does not affect the operation of any calculations based on the value being replaced. This function can also be achieved using a variable with a conditional expression, but REPLACE might be more convenient. A value set to null by

SUPPRESS DUPLICATES is not replaced with the substitution value but always prints blank.

16. The SUPPRESS DUPLICATES option prevents repetitive printing of data. For each report input record, the value of a column specified with SUPPRESS is compared to its value in the previous record. If it matches, blanks print instead of the value. If the column is part of the sort key for the main report query, the duplicate suppression does not occur on control breaks. In this context, a control break occurs when the column—or any column higher in the sort key—changes value.

    This control break behavior can be obtained for non-sort columns by using the BY phrase. The identifier after BY is an element or variable to be tested if the value of the column itself does not change. If there is a change in the value of the BY variable (or higher sort columns if the BY variable is in the sort key), suppression does not occur.

17. The QUOTE or QUOTED option surrounds the column's data with quotation marks and escapes any embedded quotation marks. This is typically used when exporting data to another program. The quote-string is a single character that is added to the beginning and end of the data. It defaults to the quotation mark (“). The escape-string is another single character whose default is the back-slash (\). If the data contains an occurrence of the quote-string or the escape-string, it is preceded in the output by the escape-string. If the quote-string and the escape-string are the same, the embedded quotes are doubled (which is the SQL convention). To specify a quotation mark, surround it by single quotes: ‘”’.

    For example, to use the single quote instead of the double quote, and to double the quote if it appears in the text, the format option would be written as:

    ```
    QUOTE "'" ESCAPE "'"
    ```

    If the original text contained the following:

    ```
    Elmer said, "That's all, folks."
    ```

    The output using the above format would be:

    ```
    'Elmer said, "That''s all, folks."'
    ```

18. The TAG option is available in HTML-style output only. It surrounds the column's data with XML opening and closing tags using the “name” string as the tag name. Additional attributes can be specified with the ATTRIB keyword followed by a name and value in parentheses. The attribute name must be a quoted string, and the attribute value may be a quoted string, a number, or a defined variable. The tag name and attributes are output according to XML standards.

    If the variable being printed in the column is an array that has more than one value, the tag surrounds the full set of values. The result is that the values are concatenated together unless a VALUETAG option is used to apply a tag to each individual value.

The TAG option must be used within an ELEMENT. Examples are shown in the description of ELEMENT.

19. The VALUETAG option is available in HTML-style output only. It is similar to the TAG option, except that it places tags around the individual values in the array of values for the column. If the column's variable has just one value, then TAG and VALUETAG have the same effect. If both are used on a column, the VALUETAG appears inside the TAG.

SET

Syntax

SET column_name [FROM variable] [, column_name [FROM variable]] … .

Used In

Calculation

Examples

```
SET aashind_l, aashind_r, aashind_h.

SET dbfifteenbar_r FROM db.
```

The SET statement is used in calculation scripts to store the results of a calculation back to the database. The value of the FROM variable is placed in the specified column. If the column and FROM variable have the same name, the "FROM variable" part may be omitted. You may use multiple SET statements or multiple columns in a single SET when the calculation script produces more than one result. The results are stored for each row you choose to be calculated. Rows can be modified only if they are editable and checked out by you. If the specified column contains manually entered data (flagged as "M") or data from prior to the existence of a calculation (flagged as "P"), it is not changed unless the user chooses to override (in the Calculation Manager dialog).

Values can be stored in two ways: singly or in groups. If the *column_name* is a column in the base table of the calculation, a single value is stored in each calculated row. If the source variable has more than one value, only the first value is used. If the calculated variable has a null value, a null is stored in the column.

A group of values can be stored by specifying a column in a table that is a direct child of the base table. This causes all existing child rows in the selected set to be updated. If necessary, rows are added or deleted to match the number of values in the source variable. More than one column in the child table may be given new values by using multiple SET statements or multiple columns in one SET. Care should be taken to ensure that all source variables have the same dimension; otherwise, data could be lost.

In case of ambiguity in column names, the form table.column may be used for *column_name*. The _r, _l, or _h suffix must be used if the column has modal values.

TEMPLATE

Syntax

TEMPLATE template-name [column-layout] output-specification .
template-name ⇒ name

Used In

Report

Example: Report (text)

```
TEMPLATE basic SEPARATOR "|"
AT LEFT FIELD WIDTH 8, FIELD WIDTH 50.
```

Example: Report (HTML)

```
TEMPLATE head1
ELEMENT "tr" ATTRIB ('style','border: 1px solid grey;')
FIELD TAG 'td' ATTRIB ('style', 'vertical-lign:middle;text-
align:center;background-color:gainsboro;font-
size:12;padding:10px;font-family:VERDANA'),
FIELD TAG 'td' ATTRIB ('style', 'vertical-
align:middle;text-align:center;background-
color:gainsboro;font-size:12;padding:10px;font-
family:VERDANA'),
FIELD TAG 'td' ATTRIB ('style', 'vertical-
align:middle;text-align:center;background-
color:gainsboro;font-size:12;padding:10px;font-
family:VERDANA').
```

A template describes the format of a report line without the data. Templates are not required but are useful to avoid repetitive specification of layout options. Putting the statement `USING` *`template-name`* into an output specification copies all the column layout information from the template into the output specification.

In a template, a set of column layout options can be given right after the template name. These options become the default for all columns in the template. There must be one and only one output specification in a template, which must be either an AT statement or an ELEMENT statement. This statement can contain additional column layout options, which take precedence over the template defaults. Finally, when a template is invoked with a USING statement, other layout options can be given and these options take precedence over the template. Column and line specifications are described under the SECTION statement.

In the output specification used in a template, it is possible to use a literal, variable, or element name as a value to be printed in a column. This would print the specified value whenever the

template is used. However, the keyword FIELD can also be used in place of a value. In this case, the value to be printed is not defined until it is specified in a USING statement. An output specification in a template definition may not contain USING.

WHEN

Syntax

WHEN expression DISPLAY message [parameter [, parameter] …] .

Used In

Validation

Examples

```
WHEN sum_pct > 100 DISPLAY "Percents sum to more than 100".

WHEN error DISPLAY "Error in horizon %s" hzname.
```

The WHEN statement is used in validation scripts to produce a message if an error condition is detected. The expression after WHEN is evaluated for each row to be validated, and if a True (non-zero) value is found, the message is added to the validation message list. If the message contains substitution markers as used in *sprintf* (such as %s or %g), values are taken from the list of parameters and placed into the message. The validation process also records information about which row generated a message, and this information is included when the message list is displayed.

In some cases, multiple values are useful for the WHEN expression, for the message, or for its parameters. This causes multiple messages to be generated for each row validated. If the validation script extracts data from a child of the base table, individual messages for each child row can be produced by using parameters that have values collected from the child rows.

Additional Information

Writing an SQL Query

Operators or Functions

Arithmetic operators, comparison operators, logical operators, and aggregate functions are used to filter the search functions of the WHERE clause. The chart on the following page identifies the data types and various comparison operators. Further information on the various operators and functions is provided as they are described in this document.

NASIS 7 uses a national database (server), a local database (client), and a selected set (screen). Queries and reports run off either the national database or the local database. Queries that are run against the national database require an Object table (e.g. Legend, Mapunit, Datamapunit). Queries written for a local database can be written to retrieve data from child tables (e.g. Correlation, Component, Horizon, etc.).

Data Types and Comparison Operators

The data types (integer, character, etc.) and comparison operators (LIKE, "=", ">", "is null", etc.) are used to establish query conditions. There is a relationship between the comparison operators and data types that must be understood. When a query is written to specify a condition in the FROM and WHERE clause, there must be a comparison operator (such as = or LIKE) that is compatible with the data element in the query conditions. For example, the data element "area name" is a "Variable Character" data type and the LIKE operator is valid for this data type.

IMATCHES is case insensitive, except for Area symbols and Map unit symbols. Whereas equals "=" indicates a full match but is not case sensitive. MATCHES (case sensitive) and IMATCHES (insensitive) were previously used in the SQL but now only work in DEFINE statements.

For case-sensitive queries, you must use the collate function. See the following example.

```
FROM datamapunit
INNER JOIN component BY DEFAULT
AND compname collate Latin1_General_CS_AS="CLARKSVILLE"
```

LIKE allows a string of characters with wildcards to be entered without regard to the case. Equals "=" is also insensitive, but the string must be complete without any wildcards.

In Microsoft SQL Server, each column, local variable, expression, and parameter has a related data type. The data type is an attribute that specifies the type of data (integer, character, money, etc.) that the object can hold. SQL Server supplies a set of system data types that define all of

the types of data that can be used with SQL Server. The set of system-supplied data types is shown in the following table.

Data Types and Comparison Operators

[Notes at the bottom of the table discuss the type of comparison.]

Data Type	Comparison Operator											
	=	!=	>	<	>=	<=	IS NULL	IS NOT NULL	LIKE " "	MATCHES " "	BETWEEN AND	IN ( )
Character												
Variable Character (String)												
Text (narrative text)	III	III	III	III	III	III				IV	III	III
Float	II	II	II	II	II	II			IV	IV		
Smallfloat	II	II	II	II	II	II			IV	IV		
Integer									IV	IV		
Smallint									IV	IV		
Datetime									IV	IV		
Bit (Boolean)									IV	IV		
Ordered Code (choice)									IV	IV		
Unordered Code (choice)			II	II	II	II			IV	IV	II	
Property	III	III	III	III	III	III				IV	III	III
Evaluation	III	III	III	III	III	III			IV	IV	III	III
Report	III	III	III	III	III	III				IV	III	III
Rule	III	III	III	III	III	III			IV	IV	III	III
Query	III	III	III	III	III	III				IV	III	III

Notes: Values for date and date-time must be entered in the correct format or an SQL error will result. NOT, AND, and OR operators are used to combine two conditions; they are not related to data type.

Blank.—Allowed.
II.—Allowed by query program, but results may not be meaningful.
III.—Allowed by query program, but will result in SQL error when query is executed.
IV.—Not allowed.

Character Strings

char.—Fixed-length non-Unicode character data with a maximum length of 8,000 characters.
varchar.—Variable-length non-Unicode data with a maximum length of 8,000 characters.
text.—Variable-length non-Unicode data with a maximum length of 2^31 - 1 (2,147,483,647) characters.

Integers

int.—Integer (whole number) data from -2^31 (-2,147,483,648) through 2^31 - 1 (2,147,483,647).
smallint.—Integer data from -2^15 (-32,768) through 2^15 - 1 (32,767).
tinyint.—Integer data from 0 through 255.
bit.— (Boolean) Integer data with value of either a 1 or 0.

Decimal and Numeric

decimal.—Fixed precision and scale numeric data from -10^38 +1 through 10^38 –1.
numeric.—Functionally equivalent to decimal.

Approximate Numerics

float.—Floating double precision number data with the following valid values: -1.79E + 308 through -2.23E - 308; 0; and 2.23E + 308 through 1.79E + 308.
real.— (smallfloat) Floating single precision number data with the following valid values: -3.40E + 38 through -1.18E - 38; 0; and 1.18E - 38 through 3.40E + 38.

If you want to use a comparison operator on a floating value, multiple by 100, convert to an integer, and match and then divide by 100 to change it back to a floating value.

Date and Time

datetime.—Date and time data from January 1, 1753, through December 31, 9999, with an accuracy of three-hundredths of a second, or 3.33 milliseconds.
smalldatetime.—Date and time data from January 1, 1900, through June 6, 2079, with an accuracy of one minute.

Examples Comparison Operators Used in an SQL Query

LIKE

```
WHERE muname LIKE "Menfro%"
```

Equal to text

```
legendsuituse = "current wherever mapped"
```

not equal

```
mustatus != "additional"
```

equal to code value

```
repdmu = 1
```

Between two values

```
muacres between ? AND ? muacres between 10 AND 50
```

greater AND less than

```
muacres >2000
muacress <5
```

Below are comparison charts for different operators showing what the operators return.

This table shows the outcome when you compare true and false values with the AND operator.

	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

The following table shows the results of the OR operator.

	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Arithmetic Operators

Operator	Meaning
+ (Add)	Addition
- (Subtract)	Subtraction
* (Multiply)	Multiplication
/ (Divide)	Division
% (Modulo)	Returns the integer remainder of a division. For example, 12 % 5 = 2 because the remainder of 12 divided by 5 is 2.

Comparison Operators

Operator	Meaning
= (Equals)	Equal to
> (Greater Than)	Greater than
< (Less Than)	Less than
>= (Greater Than or Equal To)	Greater than or equal to
<= (Less Than or Equal To)	Less than or equal to
<> (Not Equal To)	Not equal to
!= (Not Equal To)	Not equal to (not SQL-92 standard)
!< (Not Less Than)	Not less than (not SQL-92 standard)
!> (Not Greater Than)	Not greater than (not SQL-92 standard)

Logical Operators

Operator	Meaning
ALL	True if all of a set of comparisons are true.
AND	True if both Boolean expressions are true.
ANY	True if any one of a set of comparisons are true.
BETWEEN	True if the operand is within a range.
EXISTS	True if a subquery contains any rows.
IN	True if the operand is equal to one of a list of expressions.
LIKE	True if the operand matches a pattern.
NOT	Reverses the value of any other Boolean operator.
OR	True if either Boolean expression is true.
SOME	True if some of a set of comparisons are true.

Operator Precedence

Level	Operators
1	~ (Bitwise NOT)
2	* (Multiply), / (Division), % (Modulo)
3	+ (Positive), - (Negative), + (Add), (+ Concatenate), - (Subtract), & (Bitwise AND)
4	=, >, <, >=, <=, <>, !=, !>, !< (Comparison operators)
5	^ (Bitwise Exclusive OR), | (Bitwise OR)
6	NOT
7	AND
8	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
9	= (Assignment)

Wildcard Characters

Wildcard characters that can be used with the operator LIKE in the SQL to search for data. They are used to substitute one or more characters when searching for data. The standard NASIS wildcards are the underscore "_" for a single character and the percent sign "%" for multiple characters.

Other characters that are allowed are shown in the following table.

Wildcard	Description
%	A substitute for zero or more characters
_	A substitute for exactly one character
[*charlist*]	Any single character in charlist
[^*charlist*] or [!*charlist*]	Any single character not in charlist

The bracket "[ ]" wildcard uses a specific set of characters. It can be a continuous list, for example "[a-d]" (which will select any letter between a and d) or individual characters "[a,c,d]" (which will select only the three letters identified in the bracket).

The symbols "^" and "!" can be used to negate a set; e.g., [!MO123] or [^TN101] would prevent MO123 or TN101 from being selected in an area symbol query.

An ESCAPE clause can be used to search for characters that are used as wildcards. Any value can be used as an ESCAPE value; it is defined by the ESCAPE clause.

Example	Mapunit name
All map units with Menfro in the name	%menfro%
All map units with 3 to 8 percent slope in the map unit name	%3 to 8%
All map units with flooded in the map unit name	%flooded*%
All map units with Menfro and the texture silt loam in the map unit name	%menfro % silt loam%
All map units with a slope range of teens on the high end	%1_ slope%
All map units with "men" in the name and one letter before and after the three letters	_men_ %
All map units that start with CA in the name	[CA]%
All map units except the ones that that start with BC in the map unit name	![BC]%
All map units with an underscore in the name	%!_% ESCAPE !

Avoid using wildcards at the beginning of the search pattern. Search patterns that begin with wildcards are the slowest to process. Pay careful attention to the placement of the wildcard symbols because the data returned may not be what was expected.

Details on the CVIR variant of SQL are in the Syntax Reference section under EXEC SQL.

Queries

The key features of a Query script are:

- All the tables listed in the FROM clause are candidates to be *target tables* when the Query is run. For each target table picked, an SQL query is constructed to select rows in that table. Additional rows linked to the target table are then found to fill out the selected set or download list.
- Query parameters can be specified in the FROM and WHERE clause by using a comparison with a question mark; for example, "areasymbol = ?". When the query is run, NASIS creates a field for the user to enter an area symbol. It automatically looks up "areasymbol" in the metadata to determine its data type and whether it has a fixed choice list (domain).

The purpose for a NASIS query is to load the local database and to populate the selected set with data that is filtered to meet the needs of the user. The NASIS "query" requires knowledge of SQL and the database structure. The NASIS query uses two Keywords: the FROM clause and the WHERE clause. The SELECT clause is not used because the NASIS query is designed to return the data for all the columns within the table(s) identified in the FROM clause. Because queries are understood to pull all columns, the SELECT * (all columns) has been coded into the Query editor. (Do not use this syntax in your SQL).

Queries are created to retrieve data from the National Database and used again to populate the Selected Set.

A query used to "Run against the National Database" requires an Object table. The Object Table is the Parent table within an object. For example, the legend table is the object table for the Legend Object.

Queries designed to "Run against the Local Database" do not require an Object Table.

A simple query would be to load all instances of a specific map unit name. Opening the "Tables and Columns" report allows you to identify the columns, table, and data types necessary to write the query.

Table Physical Name: mapunit
Table Label: Mapunit

Seq	Column Physical Name	Column Label	Logical Data Type	Physical Data Type	Not Null?	Size
1	muname	Mapunit Name	String	Varchar	No	240
2	muname_s	S	Integer	Smallint	No	
3	mukind	Kind	Choice	Smallint	No	
4	nationalmusym	National Mapunit Symbol	String	Varchar	Yes	6
5	mapunitlfw_l	Low	Integer	Smallint	No	
6	mapunitlfw_r	RV	Integer	Smallint	No	
7	mapunitlfw_h	High	Integer	Smallint	No	

The physical name for the field map unit name is “muname” and appears in the Mapunit table. The Mapunit table is the Object table. The muname field is a string data type and is compatible with all comparison operators. An example of a simple query to extract map units from the National database follows.

Example

Load all instances of a named component into the Selected Set

```
FROM mapunit
WHERE muname LIKE “Voca sandy loam, 1 to 3 percent slopes”
```

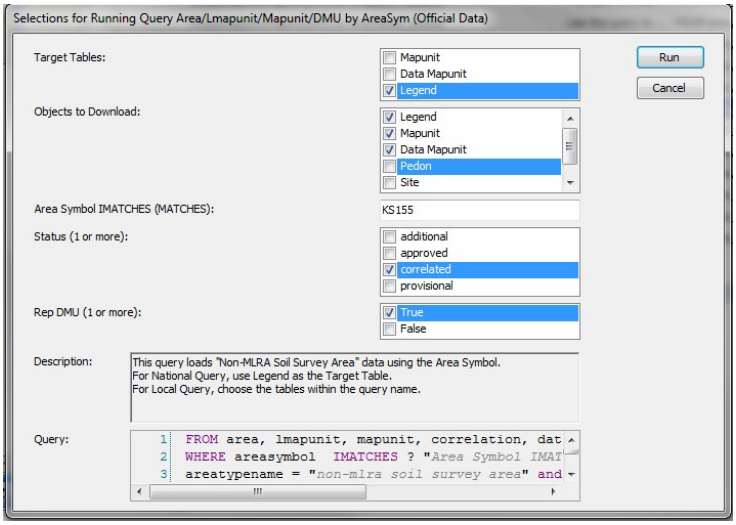
The process would be:

1. Select the Queries Explorer panel,
2. Choose to Open a new Query,
3. Enter a Query name,
4. Select the Query tab, and
5. Enter the SQL statement.

NASIS 7.3 has added enhancements to the Query function. The Query and Report “tables” option allows the user to query for and manage Queries and Reports in a table format.

Another enhancement improves how the user decides which objects to download when populating the Local Database.

The new National Query parameter box appears. The box includes the Description and Query panels. It also includes a new panel for “Objects to Download”, which allows the user to select the various objects to be downloaded from the national server.



Assuming a simple local database query to load all instances of a particular component name, then the first step in writing the query is to review the "Tables and Columns" report.

The component name column (Physical Name is "compname") is found in the Component table and the field is a variable character (Varchar).

Table Physical Name: component
Table Label: Component

Seq	Column Physical Name	Column Label	Logical Data Type	Physical Data Type	Not Null?
1	dmuiidref	Lineage	Integer	Int	Yes
2	seqnum	Seq	Integer	Smallint	No
3	comppct_l	Low	Integer	Smallint	No
4	comppct_r	RV	Integer	Smallint	No
5	comppct_h	High	Integer	Smallint	No
6	compname	Component Name	String	Varchar	No
7	compname_s	S	Integer	Smallint	No
8	localphase	Local Phase	String	Varchar	No

In NASIS, click on the "Add New Query" icon  . The General tab appears. Populate the query name and the description. Both fields are required.

The Query tab is used to write the SQL. "SELECT *" (SELECT all columns) is understood for all queries; therefore, the query begins with the keyword FROM.

A Query that has two tables and a JOIN statement with the condition in the FROM clause returns data faster than a single table with a condition in the WHERE clause. Just make sure that the second table is one that has to exist, like NASIS site, or you will lose data.

Write:

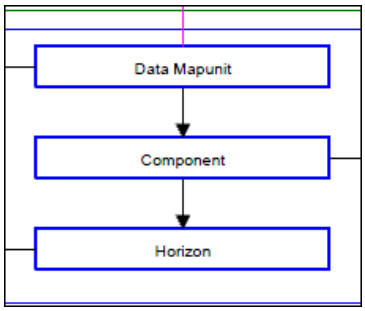
```
FROM mapunit
INNER JOIN nasissite BY DEFAULT AND muname LIKE
'relfe'
```

Instead of:

```
FROM mapunit
WHERE muname LIKE 'Relfe'
```

Target Tables

The target table focuses the outcome of a particular query. In this way, the user can control the query so that it loads only the specific data to be worked on during that editing session. The target table can greatly restrict or expand the number of records returned by a particular query. To understand target tables, the user must understand the relationship between objects in the NASIS database. The following data model diagram helps to visualize this relationship.

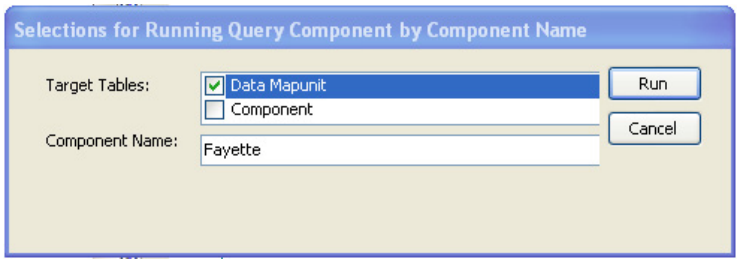


A target table restrict the records returned by a query as illustrated in the following example. In an edit session, if the user wanted to work only with components that are named "Fayette," the user would choose a query that loads components by *compname* and specify Fayette. Because component name is in the component table, either datamapunit or component could be selected as the target table.

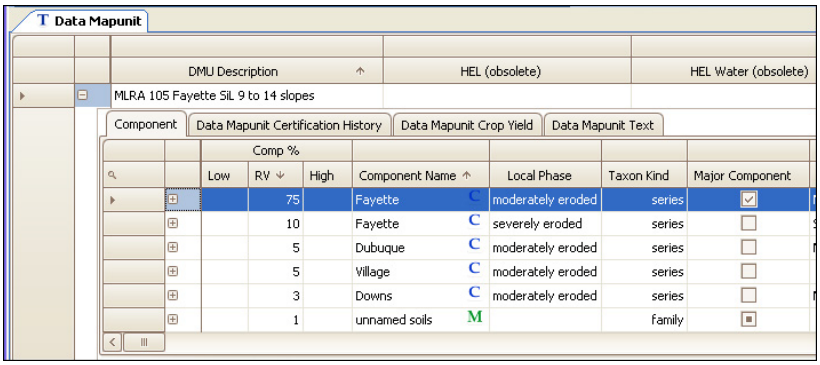
Note: Whether only the Fayette series is loaded depends on the choice of target table.

- If *datamapunit* is selected as the target table, all components of all data map units that have at least one *Fayette* component are loaded into the component table.
- If *component table* is selected as the target table, only components named Fayette are loaded.

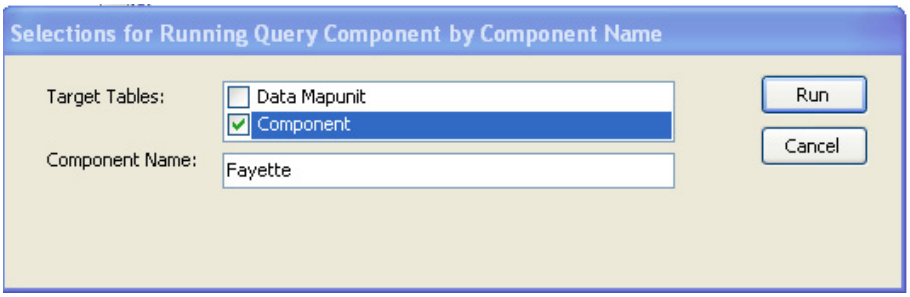
The following graphics illustrate a simple query. The query has two tables in the FROM clause that become "Target Tables" in the parameter box. In the first graphic, the target table is set to "Data Mapunit."



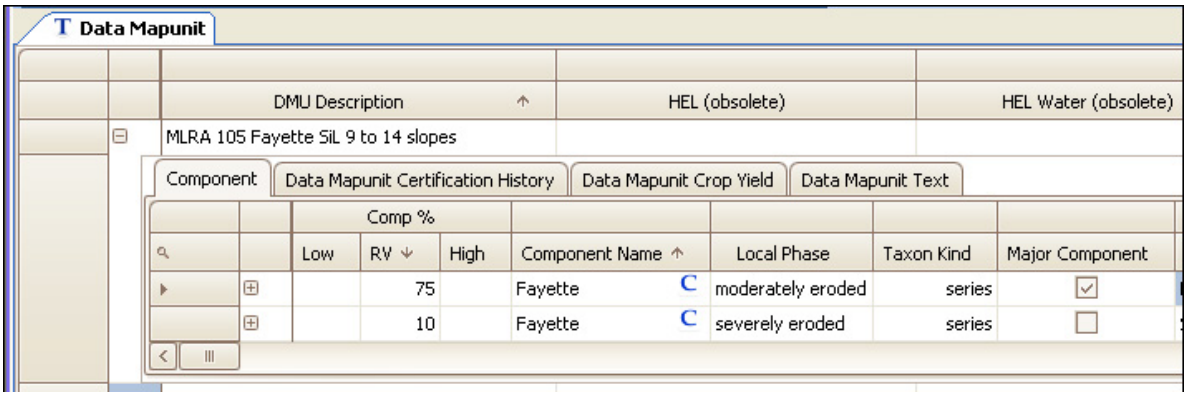
The next graphic shows that this choice provides a selected set that contains all components for data map units in which Fayette is a member.



The next graphic shows the target table changed to Component. The results will contrast with the previous results.



The next graphic show that, by using Component as the Target Table, the component table is populated with only the "Fayette" components.



The query was the same except for the target table. Changing the target table changed the results.

Setting the target table to Data Mapunit resulted in the query requesting all data map units in which Fayette is a component. A data map unit consists of all the components and their data for a given map unit.

Setting the target table to Component resulted in the query requesting that the Component table be populated with only the Fayette components. The Component table contains only a specific component; therefore, if it is used as the target table, then only the specific named component is populated within the data map unit for the selected set.

Joining Tables

SQL Express 2016 allows for a new method of joining tables and makes the queries and report more efficient. If joins are performed in the WHERE clause, the query must make a large concatenated table of all the tables in the FROM clause and then reduce the data with the joins. Note that you can put up to 256 tables in one FROM clause. The new syntax makes the first join and then pass the matching values on to the next join, reducing the size of the file.

Tables can be joined:

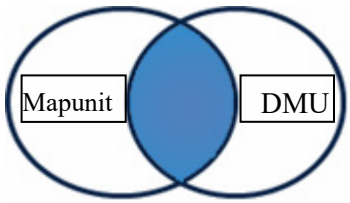
- by the relationship, commonly this uses the term "default";
- by a defined relationship, e.g., "mlra_sso" or "nonmlra_ssa" for area type joins;
- on a specific relationship, e.g., primary key:foreign key; and
- on any two fields with corresponding values.

Use the highest-level table needed for the query and work down by joining the tables. Use this method because the highest-level table has the smallest number of rows, unless one of the child tables has a restriction that reduces the number of records.

If you use the BY condition, you specify a relationship name that is defined in the NASIS data dictionary. In most cases, the relationship name is "BY DEFAULT". If more than one relationship exists between a pair of tables, you must use the correct name. The Info page (found by clicking the Blue circle icon) for a table in NASIS lists the relationship names.

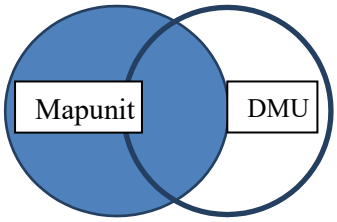
The colored area in the circle diagrams below identifies the data that is selected between the two tables in the join statement.

- INNER JOIN
  - Includes only matching values from both tables.
  - Is the most common type of join.
  - Allows you to join multiple tables in one query, but it requires a specific condition.
  - Requires that you ensure that the JOIN statement has two tables with at least one common overlapping field.
  - Is the default join type. If the type is omitted from the join clause of a query, the NASIS SQL server assumes an inner join.

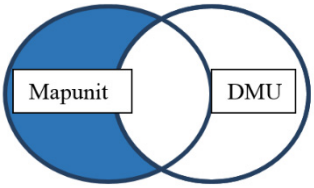


If you understand inner joins, understanding OUTER JOINS is an easy progression. They both look for and display every match they find between two tables. Both joins require that you specify the matching field(s) in the ON clause. Outer joins show the records that inner joins omit.

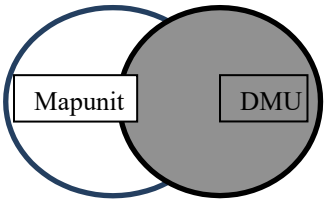
- LEFT OUTER JOINS
  - Includes all values from the left table (the table left of the word "JOIN") and only the matching values in the right table (the table right of the word "JOIN").



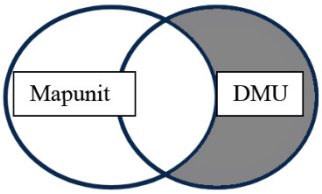
- LEFT OUTER JOIN with null values
  - Includes all values from the left table that do not match the right table.
  - Can be used instead of not exists for left table.



- RIGHT OUTER JOIN
  - Includes all values from the right table and only the matching values from the left table.



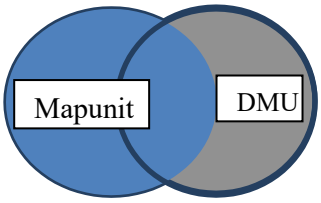
- RIGHT OUTER JOIN with null values in left table
  - Includes all values in the right table that do not match the left table.
  - Can be used instead of not exists for right table.



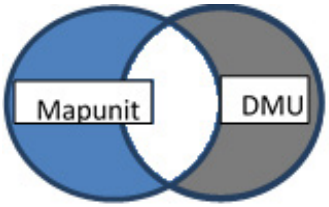
Facts to remember about OUTER JOINs:

1. Outer joins should be used only when necessary. If it is possible (i.e., the data model and business data allow), then an INNER JOIN should be used. INNER JOIN offers greater flexibility for the optimizer and doesn't mislead people into thinking that some rows of one table cannot be joined to other table.
2. Unlike in an INNER JOIN, it is important if the condition is written as join condition or in a WHERE clause. See example below.
3. Once an OUTER JOIN is used in a series of JOIN statements, the OUTER JOIN must be used on all joins below the first OUTER JOIN.

- FULL OUTER JOIN
  - Includes all values from both tables regardless of matching values.
  - Has null values in fields that lack a matching row.



- FULL OUTER JOIN with null values
  - Includes all values from both tables that do not link to the other table.
  - Is the opposite of an INNER JOIN.
  - Can also be used instead of not exists for both tables.



- CROSS JOIN (Cartesian join)
    - Creates a concatenated table from both tables. This can be handy for finding missing data that should be applied to all values in one table.
    - Is used quite rarely. Can create large number of rows returned. Relatively small tables can get quite monstrous. Some of the scenarios in which it could be used are:
        - To find all possible row combinations of some tables. Mostly, this is useful for reports where one needs to generate all combinations. For example, all taxonomic great groups by all suborders for a taxonomic soil order.
        - To join a table with just one row. Mostly, this is used to get some configuration parameters.
    - Other facts to remember about cross joins:
        - Every other join type can degrade to a cross join if the join condition is always true for all records. This is true for natural joins, inner joins, outer joins, and others.
        - To make a CROSS JOIN as a SELF JOIN, you must use table aliases so the data for each table can be identified.

- Not-EQUI-JOIN or (THETA) join
  Join conditions can have operators other than equivalence. It is only rarely used in practice; it is more just like a scientific possibility. One of the reasons it is not used is because many people cannot imagine such a possibility and therefore overlook cases it would be possible. There isn't special syntax for these joins, except that the join condition must be explicitly used. For example, you can query for map unit symbols that are the same but have different map unit names.

- SELF-JOIN
    - This is joining a table to itself.
    - When performing a self-join, a table alias must be used to make each table unique.

The relationship name is identified in the related parent table in the relationship Name column found with the blue I button in NASIS.

In general:

1. Use subqueries when you need to compare aggregates to other values.
2. Use joins when you are displaying results from multiple tables.

Write:

```
FROM mapunit
INNER JOIN correlation BY DEFAULT
INNER JOIN datamapunit BY DEFAULT;.
```

Instead of:

```
FROM mapunit, correlation, datamapunit
WHERE join mapunit to correlation
AND join correlation to datamapunit;.
```

Be careful when mixing INNER JOIN and OUTER JOIN; hunting down unintended results and side effects can be tricky. You can use parenthesis to express your logical order of precedence in the same way you'd use parenthesis in a math equation or Boolean equation. Parenthesis can make your code cleaner and more readable, and you can ensure that you get back the exact results you intended.

Below are three examples that query the same data but change the order of precedence.

1) In this query, an INNER JOIN is used after an OUTER JOIN. The query returns 483,798 records

```
EXEC SQL
SELECT count(projectiid) as pcount
FROM project
LEFT OUTER JOIN projectmapunit ON
projectmapunit.projectiidref=project.projectiid
INNER JOIN projecttype ON
projecttype.projecttypeiid=project.projecttypeiidref;.
```

2) Here, the same data is queried using both LEFT OUTER JOINS. This query returns 914,938 records

```
EXEC SQL
SELECT COUNT(projectiid) AS pcount
FROM project
LEFT OUTER JOIN projectmapunit ON
projectmapunit.projectiidref=project.projectiid
LEFT OUTER JOIN projecttype ON
rojecttype.projecttypeiid=project.projecttypeiidref;.
```

3) Here, an INNER JOIN is used inside parenthesis after a LEFT OUTER JOIN. This query returns 475,584 records.

```
EXEC SQL
SELECT count(projectiid) as pcount
FROM projectmapunit
LEFT OUTER JOIN
     (project INNER JOIN projecttype ON
     projecttype.projecttypeiid=project.projecttypeiidref)
     ON projectmapunit.projectiidref=project.projectiid;.
```

Data restrictions can be added in the "FROM" clause using the term ON. When the join conditions begin with ON, standard SQL syntax applies. You must specify the exact columns to be matched in each of the tables.

Example

```
FROM datamapunit
INNER JOIN component ON datamapunit.dmuiid=
component.dmuiidref
```

You can also add more conditions beyond just the key columns with the term AND.

Example

```
FROM component
INNER JOIN chorizon ON component.coiid=chorizon.coiidref
AND hzname LIKE "%R%" AND hzdept_r !=0
```

Joining data to a restrictive subquery causes the script to run quicker because the data set is reduced (subqueries run first, thus reducing the data set size). Join the most restrictive tables first. By applying the join in the FROM clause, you restrict the data from the start and thus decrease the amount of data being queried.

Note that other clauses for SQL statements (e.g., GROUP BY, HAVING, ORDER BY) are usable for SQL statements with joins.

Be cautious when using join conditions with columns without NOT NULL constraint. Comparing NULL values with different explicit values or even Nulls can be counter-intuitive.

The position of additional conditions is irrelevant for INNER joins for each join. However, placing the table with the most restrictive condition first in the FROM Clause is faster because it restricts the number of rows past to the next JOIN.

If the conditions are in the WHERE clause, the entire set of tables must be joined into one overly large table and then rows are removed by the conditions. If the conditions are in the FROM clause, the JOIN conditions reduce the number of rows considered with the next JOIN statement.

Tables cannot be joined directly on ntext, text, or image columns. However, tables can be joined indirectly on ntext, text, or image columns by using SUBSTRING.

Example

```
SELECT muname
FROM mapunit AS m1
INNER JOIN mapunit AS m2
```

```
ON SUBSTRING(m1.muname, 1, 20) = SUBSTRING(m2. muname, 1,
20)
```

The example above performs a two-table inner join on the first 20 characters of each text column in tables m1 and m2.

Another possibility for comparing ntext or text columns from two tables is to compare the lengths of the columns with a WHERE clause. This option is shown in the example below.

Example

```
FROM datamapunit AS dmu1
INNER JOIN datamapunit AS dmu2
WHERE DATALENGTH(dmu1.desc) = DATALENGTH(dmu2.desc)
```

Join Examples

The following query retrieves values that match in both tables. If a legend is not linked to an area, it will not be retrieved.

```
FROM area
INNER JOIN legend BY DEFAULT
```

The following query retrieves all values in the left table (correlation table) and only matching values in the right table (data map unit table). This query retrieves all map units and only data map units that are linked to the correlation table. Any data map units that are not linked to a map unit are not retrieved.

```
FROM mapunit
INNER JOIN correlation
LEFT OUTER JOIN datamapunit BY DEFAULT
```

The following query retrieves all values in right table (data map unit table) and only matching values in left table (map unit table). It retrieves all data map units and only those map units where the map units are linked.

```
FROM mapunit
INNER JOIN correlation
RIGHT OUTER JOIN datamapunit BY DEFAULT
```

The following query retrieves all values in both tables. It retrieves all map units and data map units, even if they are not linked.

```
FROM mapunit
INNER JOIN correlation
FULL OUTER JOIN datamapunit BY DEFAULT
```

The following query joins on a specific primary key (legend.areiidref) and foreign key field (area.areaiid).

```
FROM area
INNER JOIN legend ON legend.areiidref=area.areaiid
```

The following query joins a codename comparison (project.stateresponsible) and alias the table name (st).

```
FROM legend
INNER JOIN area st on
CODENAME(project.stateresponsible)=Arkansas
```

The following query performs a double join on two values (atdbiidref and areatypename) with alias for areatype.

```
FROM AREA
INNER JOIN areatype stt BY DEFAULT AND stt.atdbiidref=1 AND
stt.areatypename = "State or Territory"
```

The following query creates a "parameter" for state code and matches it to part of the area symbol. The Parameter statement is discussed later.

```
PARAMETER areasym ELEMENT area.areasymbol PROMPT "State
Symbol". SELECT LEFT((areasymbol), 2) imatches areasym
FROM legend
INNER JOIN area on areasymbol on areasymbol=areasym
```

The following query creates a double self join (dmuiidref=c2.dmuiidref AND c1.coiid!=c2.coiid). It finds duplicate components that have the same name Captina in the same data map unit.

```
FROM datamapunit
INNER JOIN component AS c1 on
c1.dmuiidref=datamapunit.dmuiid
INNER JOIN component AS c2 on
c2.dmuiidref=datamapunit.dmuiid AND c1.coiid!=c2.coiid
WHERE compname IN ("Captina")
```

CASE, WHEN, THEN, ELSE Statement

Case, when, then, else statements can be used to further reduce your selected set in reports.

- CASE.—When a condition is met, the value is TRUE.
- THEN.—Selects the output.
- ELSE.—The alternative output is selected when the CASE is FALSE.
- END AS.—Saves the output as an alias.

Example

If the subclass Land category class is empty, then just use the main class; otherwise, concatenate the class and subclass together and label it Non_irrigated_land_class.

```
CASE WHEN component.nirrcapscl is null
THEN component.nirrcapcl
ELSE component.nirrcapcl + component.nirrcapscl
END AS Non_irrigated_land_class
```

Subquery

A Subquery, also called INNER QUERY, INNER SELECT, and SUB-SELECT, can be written in the SELECT clause, FROM clause, and WHERE clause. Ensure that multiple subqueries are in the most efficient order. There are 3 types of subqueries: in-line view query, correlated subquery, and noncorrelated query.

An in-line view query (Derived Table or into temp tables) is a subquery followed by SELECT and FROM clauses.

A subquery is a query in a query. A subquery is typically added in the WHERE clause of the SQL statement, but it can also be in the FROM clause. In most cases, a subquery is used when you know how to search for a value using a SELECT statement but you do not know the exact value. Subqueries are an alternative method of returning data from multiple tables. A subquery further restricts the results of the main query. The most common use for subqueries is filtering data in the WHERE clause of a SQL command. A subquery is set within the query using parentheses. Four special operators (EXISTS, IN, ALL, ANY) and the conventional operators (such as =, <, >, >=, and <=) are used to connect the containing command and the subquery.

Use SQL JOINs instead of subqueries if possible.

Use "UNION ALL" between queries. The following example selects the map unit symbols from the two different legends and combines them into one list.

```
SELECT musym
From area
Where areasym IN (MO207)

UNION ALL

SELECT musym
From area
Where areasym IN (MO103);.
```

The mathematical operators that can be used are: scalar_expression { = | <> | != | > | >= | !> | < | <= | !< } ALL (subquery).

Subqueries Using the = Operator

What if it was necessary to identify all the component(s) with the maximum percentage in a survey area or to identify the maximum in a data map unit? The subquery using the = operator returns one result. This subquery, which extracts the maximum component percentage, is set apart using parentheses and uses that value for comppct_r. What is returned for that value and used in the main query depends on whether the subquery is "correlated" to the main query. This introduces the concept of a correlated versus uncorrelated subquery.

Subqueries Using the EXISTS Operator

The EXISTS operator tests for existence of data. The data either does exist (TRUE) or does not exist (FALSE). Therefore only one column or the asterisk (*) is necessary in the SELECT statement.

Consider the following example in which a query loads all components that have more than one texture in the surface horizon. The query prompts for a survey area and it selects the surface horizon. The subquery begins with EXISTS and tests for the existence of more than one record ID (chiidref) in the horizon texture group table for the surface layer. Notice the use of the chorizon table, which links the subquery to the query.

```
FROM areatype
INNER JOIN area BY DEFAULT
INNER JOIN legend BY DEFAULT
INNER JOIN lmapunit BY DEFAULT
INNER JOIN mapunit BY DEFAULT
INNER JOIN correlation BY DEFAULT
INNER JOIN datamapunit BY DEFAULT
INNER JOIN component BY DEFAULT
INNER JOIN chorizon BY DEFAULT
WHERE areasymbol LIKE ? AND hzdept_r = 0
AND EXISTS (SELECT chorizon_iid_ref FROM chtexturegrp
WHERE JOIN chorizon TO chtexturegrp GROUP BY
chorizon_iid_ref
HAVING COUNT(*) > 1)
```

Subqueries Using the NOT EXISTS Operator

Contrary to EXISTS, the NOT EXISTS operator identifies the nonexistence of data. For example, it could identify those components in which no horizon information is entered. Although OUTER JOIN is helpful, another method is available. A subquery can be helpful in identifying a child table that has no open rows.

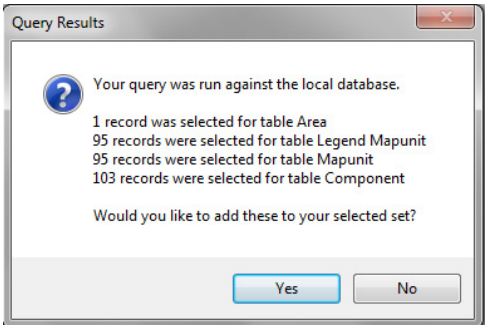
NOT EXISTS is used in the following example. The subquery is in parentheses and selects everything for the horizon table and joins the component and the horizon table. NOT EXISTS is a negative or reversal operator. If nothing exists (e.g., a table with no data), then it returns a TRUE statement and that data is loaded into the selected set.

```
FROM component
INNER JOIN chorizon BY DEFAULT
WHERE NOT EXISTS (SELECT chiid FROM chorizon
WHERE JOIN component to chorizon)
```

Correlated Subquery

A correlated subquery depends on the outer query. It uses the data obtained from the outer query in its WHERE clause. If there are millions of records, the statement with the correlated subquery is most likely less efficient than an INNER JOIN because the correlated subquery needs to run millions of times.

A correlated subquery has a more complex method of execution than single- and multiple-row subqueries and is potentially much more powerful. If a subquery references columns in the parent query, then the results are dependent on the parent query (correlated). The SQL differences, however, are subtle. Notice the subquery FROM clause in the following example and compare the two versions. In the correlated subquery, the subquery and main query are linked using the WHERE clause that links the datamapunit in the subquery to the datamapunit in the main query.




```
FROM area
INNER JOIN legend BY DEFAULT
INNER JOIN lmapunit BY DEFAULT
INNER JOIN mapunit BY DEFAULT
INNER JOIN correlation BY DEFAULT
INNER JOIN datamapunit BY DEFAULT
INNER JOIN component BY DEFAULT
WHERE areasymbol matches 'KS155' AND repdmu = 1 AND
mustatus = 'correlated' AND comppct_r = (SELECT
max(comppct_r)
FROM component
WHERE JOIN component to datamapunit)
```

In the correlated subquery, the component with the highest component percentage in each datamapunit is presented as the value to be used in the main query.

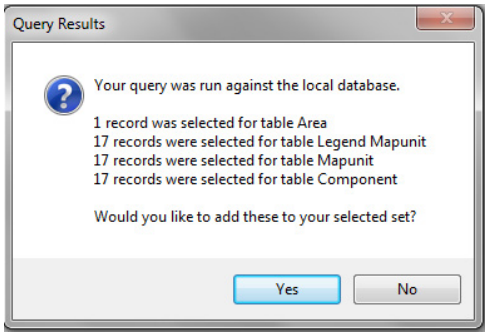
The following example shows joining subquery tables to main query.

```
FROM area
WHERE areaiid IN (select areaiidref FROM legend WHERE JOIN
legend to area)
```

Uncorrelated Subquery

A noncorrelated subquery (does not refer to the outer query) is used when dealing with large tables from which you expect a large return (many rows) and/or if the tables within the subquery do not have efficient indexes.

The following example subquery loads the component that contains that maximum percentage. Only 17 components contain the 100 percent found to be the highest comppct_r value in the survey area.




```
FROM area
INNER JOIN legend BY DEFAULT
INNER JOIN lmapunit BY DEFAULT
INNER JOIN mapunit BY DEFAULT
INNER JOIN correlation BY DEFAULT
INNER JOIN datamapunit BY DEFAULT
INNER JOIN component BY DEFAULT
WHERE areasymbol matches 'KS155' AND repdmu = 1 AND
mustatus = 'correlated' AND comppct_r = (SELECT
max(comppct_r) FROM component INNER JOIN datamapunit BY
DEFAULT)
```

Because a WHERE clause does not exist in the subquery, the extracted comppct_r results come from the query of the entire component table. A maximum comppct_r value of 100 percent is returned and presented as the value to be used in the main query.

Subqueries Using the IN Operator

The IN operator does not require that the subquery returns one value. In the following example, the IN operator identifies the maximum bottom depth of the soil. Note the use of the term MAX on the horizon bottom depth column. In addition, this query contains a second subquery to

identify those soils in which the parent material is “till”. This example illustrates the use of multiple subqueries to load data.

```
FROM area
INNER JOIN legend BY DEFAULT
INNER JOIN lmapunit BY DEFAULT
INNER JOIN mapunit BY DEFAULT
INNER JOIN correlation BY DEFAULT
INNER JOIN datamapunit BY DEFAULT
INNER JOIN component BY DEFAULT
INNER JOIN chorizon BY DEFAULT
WHERE areasymbol matches ? AND repdmu = 1 AND mustatus =
'correlated'
AND ANY(SELECT * FROM copmgrp WHERE JOIN component to
copmgrp AND pmgroupname matches '*till*' AND rvindicator =
1)
AND hzdepb_r IN (SELECT MAX(hzdepb_r)
FROM chorizon WHERE JOIN component to chorizon)
```

The ANY keyword denotes that the search condition is TRUE if the comparison is TRUE for at least one of the values that is returned. If the subquery returns no value, the search condition is FALSE. This keyword is like the IN keyword.

As a rule of thumb, if you look for many or most of the rows, try to avoid using correlated subqueries. Use a correlated subquery when the return is relatively small or other criteria are efficient; that is, if the tables within the subquery have efficient indexes.

```
SELECT DISTINCT munames
FROM mapunit
WHERE EXISTS (SELECT muiidref
     FROM correlation WHERE muiidfref is not null)
```

In the case above, the subquery runs once for each row of the main query, thus causing possible inefficiency. The case below shows the application of a join.

```
SELECT DISTINCT munames
FROM mapunit
INNER JOIN correlation on
correlation.muiidref=mapunit.muiid
```

A subquery is subject to the following restrictions:

- A subquery cannot use DISTINCT key if it includes GROUP-BY.
- A subquery cannot use COMPUTE and INTO clauses.
- A subquery can use ORDER-BY if it also has TOP().

- A subquery generated view cannot be updated.
- If a subquery returns a single value, then you can compare it using a comparison operator. If the subquery returns multiple values, you can use ANY, SOME, ALL, and IN.
- If a subquery is used with comparison operator, it must include only one column name/expression (except that EXISTS and IN operate on SELECT *)
- The COMPUTE, ORDER BY, and INTO clauses cannot be specified in a subquery.

The following example demonstrates three uses of the SELECT statement, showing how to format them. It shows using a subquery in the first SELECT of the query, assigning the result to an output column name, and a third SELECT, which assigns the result to another output column.

```
EXEC SQL
SELECT upedonid, (SELECT MAX(hzdept) FROM REAL phorizon) AS
maxdep,
(SELECT MIN(hzdepb)
     FROM REAL phorizon) AS Mindep
FROM REAL pedon;.
```

The following shows sample output from the example above.

User Pedon ID	maxdep	Mindep
99KS131001	710	1
2000KS027005	710	1
85KS061532	710	1
85KS061533	710	1
87KS013002	710	1

If you use COUNT in the SQL instead of a DEFINE statement, you need to multiply the count by a very small number to keep the value as a number; otherwise, the number transforms into a text value and does not sort correctly.

Example

```
Select COUNT(muacres) *0.00001 As "Map Unit Acres"
```

Using Subreports

The purpose of a subreport is to produce some output that is loosely coupled to the primary report. A subreport has its own set of queries and output specifications, which might not be related to those of the primary report. A subreport allows for greater flexibility in cases where complex formatting is required.

A subreport is requested with an INCLUDE statement in the data block of an output section. The entire output of a subreport is inserted in the data block as a single logical line. If keep processing is in effect, it attempts to keep the subreport output together on a single page. Therefore, it is advisable to design subreports that output less than a page. Longer output spills over onto additional pages of the main report and can produce unwanted results. It is also possible, however, to have a main report that produces no output of its own and only calls a series of subreports. In such a case, the main report is a page by page copy of the subreports.

Subreports may not specify page layout, such as the page size, font, headers, or footers. The page layout of the main report controls all output from subreports.

A report and its subreports do not need to use the same base table, and no automatic synchronization is done (as with properties in a DERIVE statement). Subreports may call themselves in a recursive fashion to produce a report on recursively organized data. An example is a report that lists rules and all their subrules at any depth. It is important to pass the correct parameters to a subreport so that it will find the correct records to report on; otherwise, the subreport might generate an endless recursion.

Parameters for Web Soil Survey Reports

A general report script can use any parameters, but reports for Web Soil Survey (WSS) must use the parameters that WSS supplies. In general, a user in WSS can choose either a full soil survey area or a set of map units to report on. These choices are passed to the report script. Other parameters are used in specific reports. The parameter names and types must be defined as shown in the list below. The report script can also define a prompt for each parameter, which appears in the user interface for entering parameter values when applicable.

```
PARAMETER areasym CHAR.
```
The area symbol for the requested survey area. If selecting by map unit, this is null.

```
PARAMETER mukeys MULTIPLE CHAR.
```
The map unit keys for the selected map units. If selecting by survey area, this list is empty.

```
PARAMETER includeminor BOOLEAN.
```
Set to 1 if minor soils are to be included in the report or to 0 if they are to be excluded.

```
PARAMETER useNationalMapunits BOOLEAN.
```
Set to 1 if national mapunit symbols (mukey) are to be printed or to 0 if survey based map unit symbols (musym) are to be used.

```
PARAMETER crops MULTIPLE CHAR.
```
The list of crop names selected for inclusion in a crop yield report.

The last example uses what is called a "dynamic choice list," meaning that the list of choices depends on the area of interest selected by the user. There is currently a limited set of parameters that can be used as dynamic choice lists. In place of `crops`, the following parameter names can be used:

- `cocrops`: Crop names from the Component Crop Yield table that have non-null yield data.
- `mucrops`: Crop names from the Mapunit Crop Yield table that have non-null yield data.
- `interps`: Names of interpretations (rules) exported with the selected survey area(s).
- `sainterps`: Same as interps but allows more than 3 selections.
- `cotextkinds`: Component text kind and category.
- `mutextkinds`: Mapunit text kind and category.
- `nontechdesccats`: Mapunit text categories where text kind is "nontechnical description".
- `counties`: County names found in the Legend Area Overlap table for the area of interest.

Using Parameters in a Report Query

Most of the parameters listed above are used in the report queries to control the selection of data. In one case, "useNationalMapunits", the parameter is also used in the body of the report to alter parts of the output. In the queries, some unusual SQL features are used. This following example from a crop yield report shows most of the parameter options.

```
1) EXEC SQL SELECT areaname, musym, museq, mapunit.mukey
     muiid, muname,
2) Nationalmusym, compname, comppct_r, localphase,
     component.cokey,
3) cropname, yldunits, irryield_r, nonirryield_r
4) FROM legend
5) INNER JOIN mapunit BY DEFAULT
6) INNER JOIN component BY DEFAULT
7) OUTER JOIN outer cocropyld BY DEFAULT
8) WHERE ( areasymbol=$areasym OR mapunit.mukey in
     ($mukeys) )
9) and ( majcompflag = "Yes" OR 1 = $includeminor )
10) and cropname in ($crops)
11) ORDER BY case when $useNationalMapunits =0 then
     areaname else nationalmusym end,
12) museq,
13) cropname;
14) AGGREGATE ROWS muiid
15) COLUMN yldunits UNIQUE GLOBAL
```

```
16) CROSSTAB cropname VALUES (crops)
17) CELLS irryield_r, nonirryield_r, yldunits.
```

Line 8 uses the "areasym" and "mukeys" parameters to select the map units for the reports. By using OR, the line selects either by survey area or by map unit, depending on which parameter is supplied. Note that when testing this report, if you enter values for both parameters, the report selects map units that meet either criterion, which may be more than you expected.

Line 9 uses the "includeminor" parameter to select minor components. Here OR means, "select a component if it is a major component OR if the includeminor parameter is set to 1."

Line 10 uses the "crops" parameter to select crop names. The parameter is used again in line 16 to specify the crosstab values. Doing this forces the report to include a column for each crop, even if there is no yield data for a crop. It also forces the columns to appear in the order that the crop names are entered in the parameter.

Line 11 makes use of the "useNationalMapunits" parameter to control the sorting of the report. This is tricky because when survey area map unit symbols are used, the report is sorted by survey area; but when national map units are used, the report is not sorted by survey area. The SQL CASE expression is used here. CASE is similar to the IF expression in a DEFINE statement. Line 11 says to sort by area name or by national musym depending on whether survey area or national symbols are used.

Line 12 says to sort on museq. The column museq is the sequence number for map units within a legend, so it is used to sort when using survey area map units. It has no effect when using national map unit symbols (and does no harm).

Script Variables

A script variable is a special notation used to obtain information about the report script itself from the database. An example is the name of the report script, which is stored in the "report" table in the column "report_name". To include this name in the output of a report, use a reference to the script variable SCRIPT_NAME, as in:

```
DEFINE reportname INITIAL SCRIPT(SCRIPT_NAME).
```

This places the script variable into a normal report variable, reportname, which can then be printed as part of the report heading. Using INITIAL in this statement means that the variable *reportname* is set at the beginning of report processing and never changes.

The script variables available in NASIS are a little different from the ones in WSS (from the Soil Data Mart database) due to differences in table structure. The script variables are:

SCRIPT_NAME
The name of the report.

REPORT_TITLE
In NASIS, this is the same as SCRIPT_NAME. In WSS, this is the report title from the Home tab in the report editor.

REPORT_HEADER
In NASIS, this is blank. In WSS, this is the text used as a headnote in the report from the Home tab in the report editor.

INTERP_NAME
In NASIS, this is blank. In WSS, this is a list of rule names included in an interpretation report from the Interpretations tab in the report editor.

INTERP_TITLE
In NASIS, this is blank. In WSS, this is the list of column headings used with the corresponding rule in the INTERP_NAME list. This is also entered in the Interpretations tab.

SCRIPT_ID
The internal ID number for the record in the report table. This might be used to query for data linked to the report in the database.

Using script variables allows a report script to be generalized so that certain features do not have to be coded into the script. This capability is used for WSS interpretation reports, which use a completely generic script. All standard interpretation reports in WSS use the same script. That script uses script variables to pick up the report name, headnote, and list of interpretations from data stored in the report table. To create a State report using custom interpretations, you can simply copy an existing interpretation report and change the list of interpretations and titles in the Interpretations tab. You also specify which states will use the report in the Usage tab.

NASIS CVIR Script Writing References

Database Structure Guide

The NASIS 7.0 Database Structure Guide is a comprehensive reference that describes all aspects of the NASIS database design. The guide provides information you need to know about the NASIS template model, naming conventions, and data types. It can be obtained from the NASIS website:
https://www.nrcs.usda.gov/wps/portal/nrcs/detail/soils/survey/tools/?cid=nrcs142p2_053547.

Table Structure Report

The Table Structure Report is included in the NASIS 7.0 Database Structure Guide. The report provides information you need to know about table and column physical names, modality, data types, and other characteristics necessary for report writing.

Database Structure Diagrams

The Database Structure Diagrams are included in the NASIS 7.0 Database Structure Guide and the NASIS Online Help. Because of the size of the database, the diagrams each show just one object hierarchy. They show the table relationships required for completing joins between tables.

Suggested Reading

http://www.w3schools.com/xml/xml_whatis.asp

http://www.htmlhelp.com/reference/html40/

http://www.htmlgoodies.com/

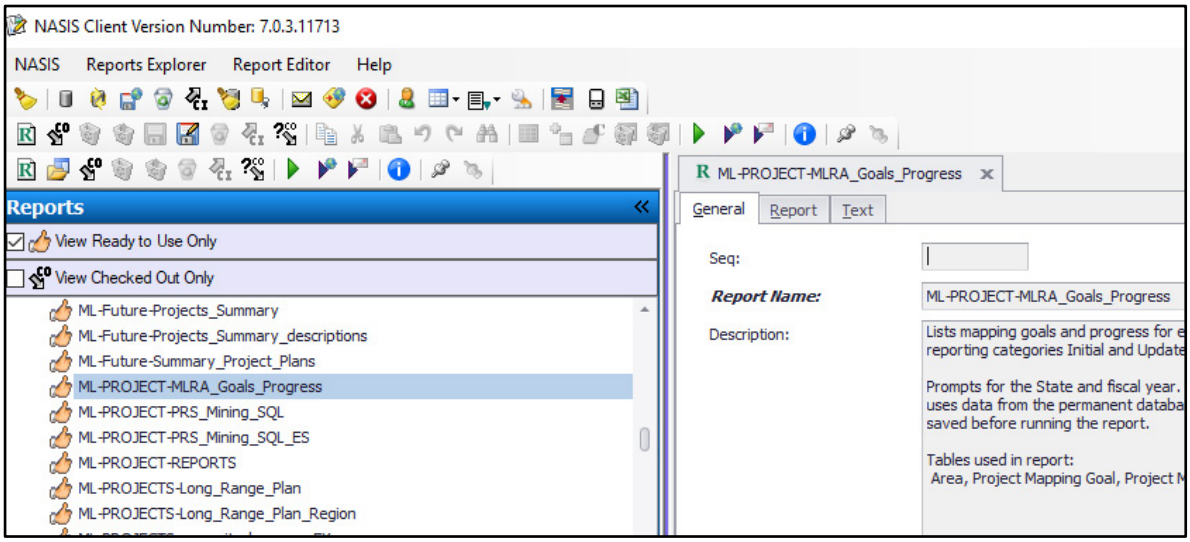
https://msdn.microsoft.com/en-us/library/ms189826(v=sql.90).aspx

Web Uniform Resource Locator (URL) Reports

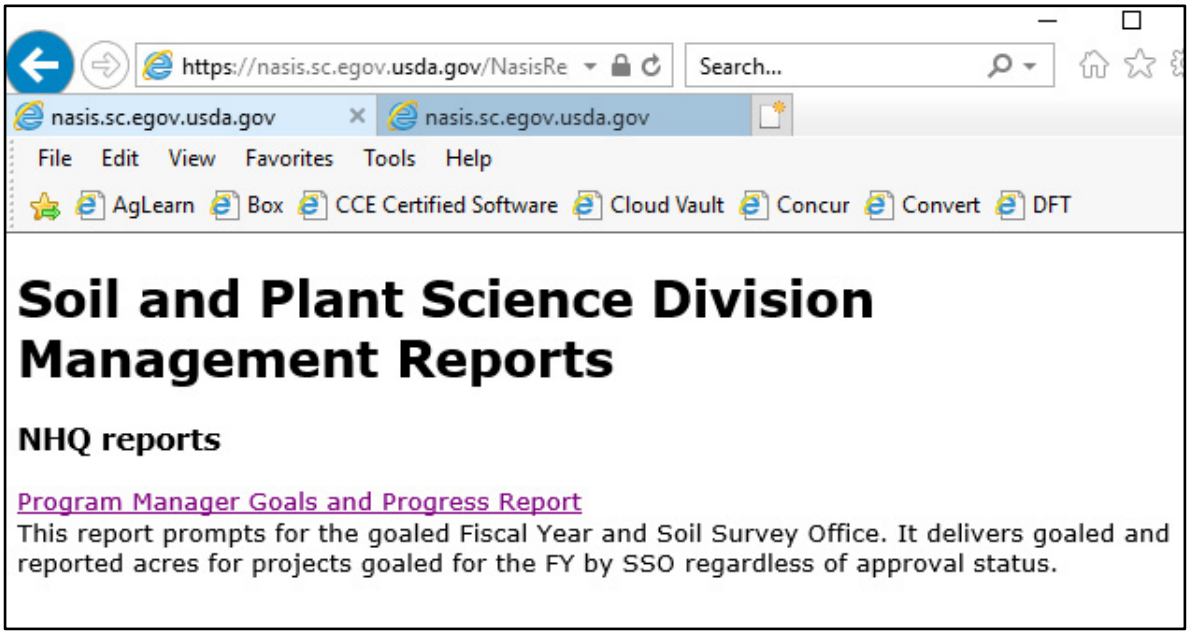
Overview

Any NASIS report, using any report format, can be called via URL as long as it is in the correct NASIS site. This function allows anyone to run reports outside of the NASIS Client, and the reports can be run on any device with a browser and network connection. See the screenshots below as an illustration of a report run in NASIS and also run via a URL.

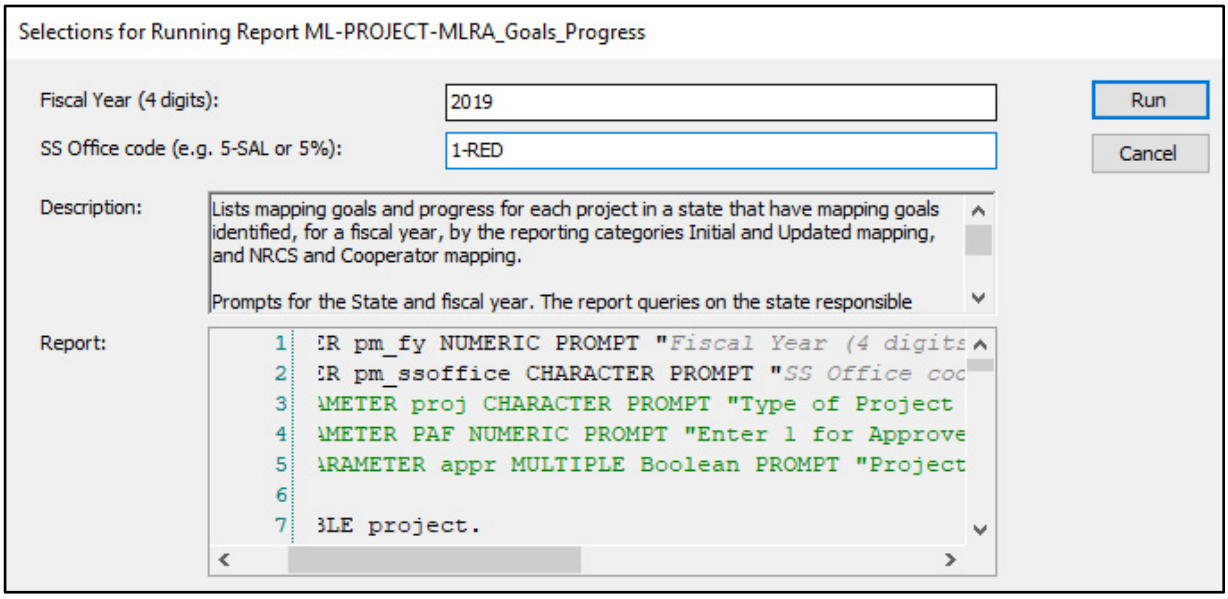
Project Manager Goals and Progress Report in NASIS



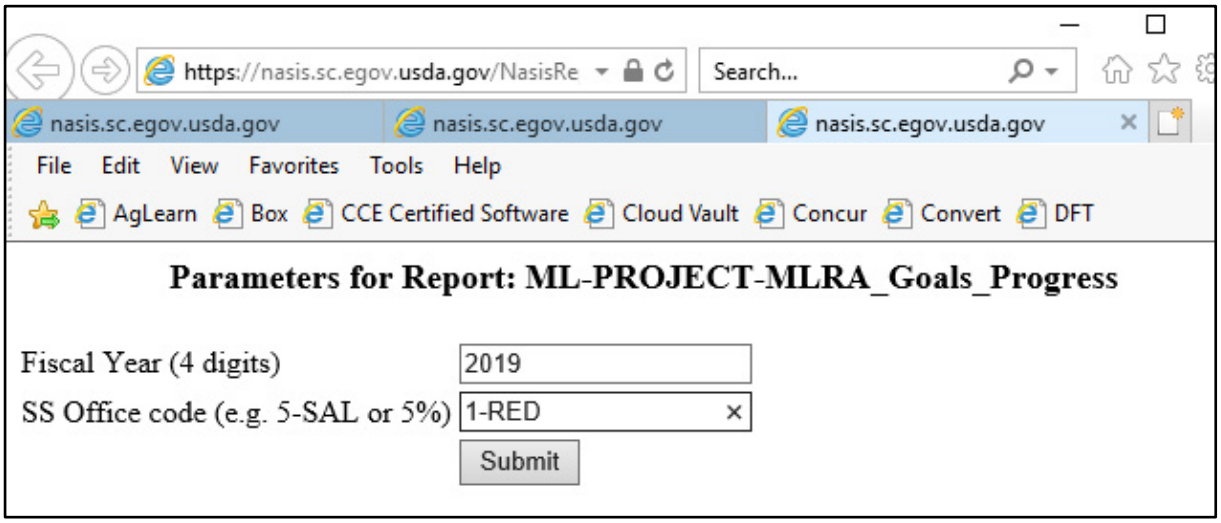
Same Report in a Web Browser



Parameters When Running the Report in NASIS



Parameters When Running the Report in a Web Browser



Following is an example NASIS report script of a URL report. For the sake of conciseness, only the parameters and initial SQL statements are included. Note how “pm_fy” and “pm_ssoffice” are passed through to the SQL in the FROM statement.

```
-----------------------------------------------------------------------------------------------------------------
PARAMETER pm_fy NUMERIC PROMPT "Enter Fiscal Year (4 digits)" REQUIRED.
PARAMETER pm_ssoffice CHARACTER PROMPT "Enter MLRA SSO Symbols (e.g. 5-SAL; Use wildcard to load entire region.  For
example, enter 5-%.)" REQUIRED.

BASE TABLE project.

EXEC SQL
SELECT
        1 as ID, fiscalyear/1 fiscalyear, a.areasymbol ssoffice, Replace(LEFT(a.areasymbol, 2), "-","" ) AS region, projecttypename, CASE
        WHEN projectapprovedflag = 1 THEN 'Yes' ELSE 'No' END approve,  projectiid, p.projectname, username, initnrcsacresg,
        initcoopacresg, updtnrcsacresg, updtcoopacresg, stateresponsible

FROM REAL area AS a
INNER JOIN project AS p ON a.areaiid = p.mlrassoareaiidref
        AND a.areasymbol LIKE pm_ssoffice
        AND projectapprovedflag=1
INNER JOIN projectmappinggoal pmg ON pmg.projectiidref = projectiid
        AND pmg.fiscalyear = pm_fy
INNER JOIN projectstaff ps ON ps.projectstaffiid = pmg.projectstaffiidref
INNER JOIN nasisuser nu ON nu.useriid = ps.projectstaffuseriidref
INNER JOIN  projecttype pt ON pt.projecttypeiid = p.projecttypeiidref;
SORT BY  region SYM, ssoffice, approve desc, projecttypename, p.projectname, projectiid
AGGREGATE ROWS ssoffice, projecttypename
COLUMN username LIST "; ", initnrcsacresg SUM, initcoopacresg SUM, updtnrcsacresg SUM, updtcoopacresg SUM.
-----------------------------------------------------------------------------------------------------------------
```

Important: Only NASIS reports owned by the KSSL NASIS Site (those stored in the KSSL folder) can be called via URL.

The base URL is as follows.

https://nasis.sc.egov.usda.gov/NasisReportsWebSite/limsreport.aspx?report_name=

To call a KSSL-owned NASIS report, the report name is added to the end after the equal sign.

https://nasis.sc.egov.usda.gov/NasisReportsWebSite/limsreport.aspx?report_name=***Goals_Progress_Sum_by_Project_Type-Region***

The URL report always runs against the national NASIS database and does not require the NASIS Client to be installed on the user's device. The results of the URL report are identical to the results of the report run against the national database within the NASIS Client.

The output of the URL report opens in web browsers and can be copied and pasted into other applications, such as Microsoft Excel and Microsoft Word. The output can also be printed to hard copy, printed to Adobe PDF, saved as an HTML file, or saved as a TXT file.

URL Report PARAMETERS

Because URL reports always run against the national database, they normally need PARAMETERS to help filter the results. Without PARAMETERS, the reports could attempt to retrieve too many records and then fail with a timeout warning. Refer to the Report Syntax section of this document for more information about PARAMETERS.

PARAMETERS in URL reports function differently for columns that have a “choice” or than for columns that have a “Boolean” logical data type. Choices, which are also referred to as domains, have a “smallint” physical data type, and Boolean have a “bit” physical data type. The URL reports only recognize PARAMETERS of the physical data type. The NASIS client reports, however, can recognize PARAMETERS using the domain codes and Boolean check boxes.

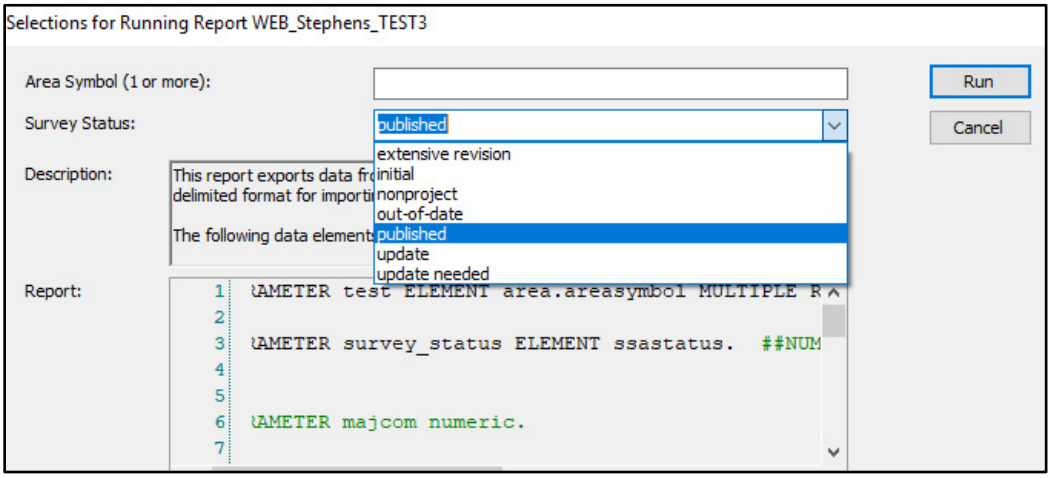
Example to Demonstrate Differences Based on Data Type

The domain for legend soil survey status is summarize below.

ID (Integer stored in NASIS)	Data Entry Text
1	out-of-date
2	published
3	nonproject
4	initial
5	extensive revision
6	update
7	updated needed

When soil survey status is included as an ELEMENT in a PARAMETER, the data entry text appears as shown below.

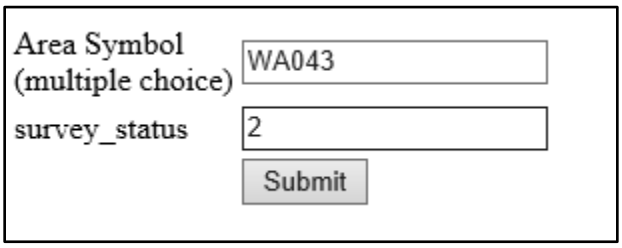
```
PARAMETER survey_status ELEMENT ssastatus.
```



The URL reports can only recognize the integers; therefore, the PARAMETER above causes the URL report to fail. The only way to include a domain as a parameter in a URL report is to use the NUMERIC value type command, as shown below.

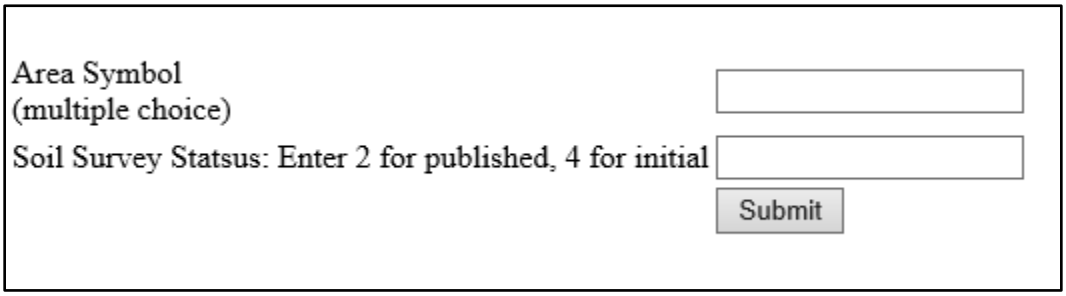
```
PARAMETER survey_status NUMERIC.
```

The PARAMETER shown above produces the following in the URL report. The user needs to know the integer values to enter, which requires a more complete understanding of NASIS domains.



As shown below, the PROMPT commands can be used to provide users with some knowledge of the integers that could be entered.

```
PARAMETER survey_status NUMERIC PROMPT "Soil Survey Statsus: Enter 2 for published, 4 for initial".
```



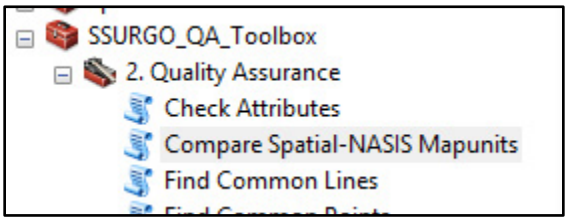
The NUMERIC function must also be used if a Boolean column is used in a PARAMETER; although, the only possible integers are 0 or 1, which equate to no or yes.

If the URL has the parameter identified with the ampersand (&), the report will run. See highlighted portion of URL below.

https://nasis.sc.egov.usda.gov/NasisReportsWebSite/limsreport.aspx?report_name=WEB-Mapunits%20by%20area%20symbol&area_sym=MO123

Calling URL Reports with Python

Web URL reports can be called with python scripts in applications such as ArcMap or ArcPro. This function allows the connection of NASIS tabular data to spatial data. For example, the SSURGO_QA_Toolbox contains a script that compares the spatial map unit symbol to the NASIS map unit symbol.



The python script calls the NASIS report "WEB-MapunitsAreasMustatus&area_sym" and uses the NASIS report output to complicate the quality assurance check. The following is a screenshot of python code.

```
# Hardcode NASIS-LIMS Report Webservice for retrieving MUSYM and MUKEY values for a specified AREASYMBOL
#
# New NASIS report that allows user to specify any of the MUSTATUS values
theURL = r"https://nasis.sc.egov.usda.gov/NasisReportsWebSite/limsreport.aspx?"
theParameters = "report_name=WEB-MapunitsAreaMustatus&area_sym="
```

Examples and Exercises

The following text is taken from page 7 of chapter 18 of the NASIS Training Materials. The materials include numerous examples and exercises and are available online at

https://www.nrcs.usda.gov/wps/portal/nrcs/detail/soils/survey/tools/?cid=nrcs142p2_053545.

> *The examples in this chapter are a sample of some approaches to writing NASIS reports. Over time, you will develop your own techniques and style. Exercises in this section build on concepts demonstrated in the examples. These exercises provide an opportunity for you to develop your own approaches to creating NASIS reports.*
>
> *When writing reports from scratch, it helps to have a report writing methodology similar to that described for writing queries in the NASIS User Guide. You may ask yourself several questions.*
>
> - *What data do I want in the report?*
> - *Are the data in the database or do they need to be calculated or decoded?*
> - *In which tables are the data?*
> - *What tables are needed to complete the joins between tables?*
> - *How do I want the data organized?*

- *How do I want the page layout to look?*

After you have defined what you want in your report, write your report using the statements and guidelines in this technical reference guide. Trial and error is almost always needed to write a report that runs cleanly. Seldom will a report be written perfect on the first try. With practice however, moderately complex reports can be written to meet a wide variety of uses.

In some cases, existing reports can be found that nearly meets the users' needs. A short-cut to writing a new report is to simply copy the existing report and modify it to meet the needs. Scan the existing report to identify parts that need to be modified and make the changes. Even with this short-cut, trial and error is needed to create a report that runs cleanly and provides the desired result.

Appendices

Appendix 1: Conventions for HTML Reports and Web Soil Survey Rule and Report Manager

DocBook XML

The Web Soil Survey Report Manager follows the guidelines in DocBook. Standard HTML coding does not work.

This appendix describes the conventions for reports used to create HTML reports like those used in the Web Soil Survey. If you want to produce a report in that style, the report script must produce XML output that conforms to the DocBook XML standard. A DocBook reference manual is available at https://tdg.docbook.org/tdg/4.5/docbook.html.

Within the broader world of DocBook applications, we are using just those features needed for Web Soil Survey reports. Because WSS reports are intended for use within a larger soil survey document, the outermost element of a report is `<section>`[1]. That `<section>` uses the attribute label="Soil Report" to identify it. Furthermore, the DocBook standard requires that every `<section>` must include at least a `<title>` element. All of these requirements can be met by including a standard AT START section in a report script, such as:

```
SECTION WHEN AT START
     DATA
          ELEMENT OPEN "section" ATTRIB ("label", "Soil
          Report").
          ELEMENT "title" report title.
END SECTION.
```

In the example above, notice the syntax of the ELEMENT command. If the word ELEMENT is followed by OPEN or CLOSE, the output contains only the opening or closing tag of the element. If not, a complete XML element with opening and closing tags is produced. If ELEMENT OPEN is used, there must be a corresponding ELEMENT CLOSE somewhere in the report script (with exceptions as noted below).

The first ELEMENT line is the beginning of the outermost section, as described above. The report language syntax requires quotes around the tag name ("section"), attribute name ("label"), and value ("Soil Report"). An element can have more than one attribute by simply adding more ATTRIB ( ) specifications.

Because this line only opens the `<section>`, there must be a closing tag as well. Normally this is handled using a SECTION WHEN AT END block in the report script, such as:

```
SECTION WHEN AT END
     DATA
          ELEMENT CLOSE "section".
END SECTION.
```

In the narrative, we use the XML convention of angle brackets to designate an element, such as <section>. In examples of report scripts, the element names are in quotes, using the report language syntax.

The line beginning ELEMENT "title" is a complete element definition and produces the required `<title>` for the `<section>`. In WSS reports, the title of the outermost section is the name of the report or table. This example prints the contents of a script variable named report title, which is explained later.

XML Elements Used in Reports

After the AT START section, the report script contains SECTION blocks to specify the contents of the report. The ELEMENT command is the main output formatting control and is used either in a TEMPLATE or directly in a SECTION.

The ELEMENT command can contain a list of data fields, each of which can have a TAG, VALUETAG, and attributes. Tags and valuetags generate complete XML elements that are placed between the beginning and ending tags of the main element. This creates a structure where the outer element can contain data or other elements. The difference between a TAG and a VALUETAG appears when displaying a report variable that has multiple values. A TAG surrounds the entire set of values for the variable, while the VALUETAG surrounds each individual value of the variable.

Some examples illustrate the way ELEMENT can be used.

ELEMENT "title" "Map unit Description".
This is the simplest case, consisting of an element name and a string of data. It translates to the following XML:

```
<title>Map unit Description</title>
```

ELEMENT "col" ATTRIB ("width", "3*").
An element can have attributes as well as data. Attributes become part of the opening tag for the element. In this case, the element has no data so the opening and closing tags get condensed into a single tag:

```
<col width="3*" />
```

ELEMENT "tr" ATTRIB ("class", "heading") FIELD TAG "td" ATTRIB ("class", "begindatagroup"), FIELD TAG "td" ATTRIB ("class", "enddatagroup").

This example shows the use of TAG to create elements within an element. Each of these elements can have attributes of their own. The first ATTRIB goes with the <tr> element, and the others go with their adjacent <td> elements.

Notice the syntax for the example. The content of an element follows the element name and optional ATTRIB. In this case, the content is two FIELDs, so the example would have to be inside a TEMPLATE. The keyword TAG follows FIELD to indicate that the field will be surrounded by tags to create a <td> element. The comma separates the two fields in the <tr> element.

If this example appeared in a TEMPLATE, there would have to be a USING statement to specify the data to be printed. This looks like a heading, so the data might be "Plant Name" and "Pct. Composition". Then the XML would be:

```
<tr class="heading"><td class="begindatagroup">Plant
Name</td>
<td class="enddatagaroup">Pct. Composition</td></tr>
```

```
ELEMENT "tr" ATTRIB ("class", shading)
     hzname TAG "td" VALUETAG "para",
     hzdept_r TAG "td" VALUETAG "para" ATTRIB ("role",
     "number").
```

This example prints data from variables that have multiple values. We can assume that the query in this report used AGGREGATE by component and specified NONE as the aggregation type for the horizon names and depths (*hzname* and *hzdept_r*). The TAG "td" places each of these variables in a <td> element, which represents a table cell. Then each individual value is placed in a <para>, which stands for paragraph and means that each appears on a separate line of output. Thus, the horizon names and depths are listed as side by side columns, although each column is actually within a table cell. This is a common style for soils reports.

Notice the attributes in the example. The <tr> has an attribute value that is not in quotes. It refers to a variable named shading, which would have been created in a DEFINE statement to hold some appropriate class attribute, such as "even" or "odd" (see table of attributes below). Also, *hzname* has no attribute; therefore, it gets the default format, which is simply text left justified in the cell.

Assuming some arbitrary data for shading, hzname, and hzdept_r, the example would create the following XML. The XML has been indented to make it more readable, but the indentation is not part of the actual report output and has no meaning.

```
<tr class="odd">
     <td>
        <para>A</para>
        <para>B</para>
```

```
          <para>C</para> </td>
        <td>
       <para class="number">0</para>
       <para class="number">15</para>
       <para class="number">75</para>
       </td>
  </tr>
```

ELEMENT "variablelist" VALUETAG "varlistentry" hzname TAG "term", hzdept_r TAG "listiem" VALUETAG "para".

This example shows the same data as the previous example in a semi-tabular format rather than in a table. It uses the DocBook element <variablelist>, which can be used to create bullets or numbered lists, depending on the final style choices. In the report script, we do not need to be concerned with the formatting details, only the structure.

A <variablelist> contains <varlistentry> elements, each of which contains a <term> and a <listitem>. The <listitem> must contain some other element, in this case <para>. To accomplish this, a VALUETAG must be at the element level. This means that the <varlistentry> element is produced for each set of values within the element. An additional VALUETAG on the *hzdept_r* produces a <para> around the horizon depth. The resulting XML is thus four levels deep.

```
<variablelist>
     <varlistentry>
          <term>A</term>
          <listitem>
               <para>0</para>
          </listitem>
     </varlistentry>

     <varlistentry>
          <term>B</term>
          <listitem>
               <para>15</para>
           </listitem>
     </varlistentry>

     <varlistentry>
          <term>C</term>
          <listitem>
               <para>75</para>
          </listitem>
     </varlistentry>
</variablelist>
```

Elements Used in Tables

The DocBook standard includes the HTML standard for creating tables, which is familiar to most people who have created webpages. The basic elements of a table are listed below. All of them are needed for a complete WSS report, but many of the elements can be left out of a report that is not for publication.

- <table> encloses the whole table.
- <title> is for a title that is printed above the table. In WSS reports, this is the name of the soil survey area. If the report uses national map unit symbols, there is no survey area and the table title is omitted.
- <col> specifies attributes for a column in the table. If used, there must be one <col> element for each column. In WSS reports, this is used to specify a relative width for the column.
- <thead> is for the table header. Rows within a <thead> receive special formatting so they look like a heading. They are also repeated at the top of each report page when the report is converted to PDF.
- <tbody> is the body of the table and occurs after the <thead>.
- <tr> is a row of a table. It can be used within a <thead> or a <tbody> and contains one or more <td> elements for the data in the row.
- <td> is table data, sometimes called a cell of a table. It must be within a <tr> element.
- <para> is a paragraph that can be included in a <td>. It is not required because data can be included within a <td> directly. The <para> element is commonly used in WSS reports to identify the type of data so that standard formatting styles can be applied.

Elements Used in Non-table Reports

Reports that are not formatted as a table include the various Map Unit Description reports, the Interpretation Description, and similar reports. Although there is no <table> element, there is an internal structure to these reports and the <section> element is generally used to represent it. The DocBook standard allows sections to be included within sections. So, for example, a report organized by survey area can have a <section> for each new area name. A Map Unit Description report can have a <section> for each map unit within the area and a variety of <section>s within the map units. Each <section> must have a <title>, which is used as a heading for the section when it is displayed.

Data in these reports is typically displayed in a list (semi-tabular) format. The DocBook <variablelist> element is used to create a list structure, which can then be displayed in various ways. The final example above provides an explanation of the use of <variablelist>.

Reports that contain large blocks of text use the <para> element to enclose the text. The way you expect the text to be stored in the database influences the attributes applied to the <para>. If the text is simply a single paragraph, then no special attributes are needed. If the text contains line breaks, tabs, or spaces that need to appear in the output, then ATTRIB ("role", "preservenewlines") is needed. If the text contains XML markup that needs to be preserved in the output, then use the SPECIAL option, as in:

```
textdata TAG "para" SPECIAL.
```

Example

This example adds a point as a bullet (special character)

```
element OPEN "h1" ATTRIB("style",p1)"&#8226;" Special.
```

Attributes Used in All Elements

Attributes describe the type of data or the role played by an element in the overall report structure. This information is used when formatting the report for display, and it can be used in different ways depending on the display format. For example, the same report might appear in a website and in a PDF document meant for printing. This separation of the functional description of a report and its display formatting allows one report script to serve multiple purposes. However, only certain attributes are recognized by the formatter, and unrecognized attributes are ignored. So the conventions listed here must be followed closely to produce reports with a consistent appearance.

Element	Attribute Name	Attribute Value	Meaning
"section"	"label"	"SoilReport"	Required to identify the outermost section of a soil report.
		"Survey_Area"	Data for one survey area. The title of the section is the survey area name. Used in reports that don't have tables.
		"Map_Unit_Description"	Data for one map unit in a map unit description report.
		others	The Map Unit Description report has labels to signify the type of data in each section, but they are currently ignored by the formatting program.

Element	Attribute Name	Attribute Value	Meaning
“title”	“role”	“suppressTitle”	Do not display a title for the current section. Although the <title> element is required in a <section>, this will suppress display of the title.
“table”	“orient”	“land”	Table is wide and should be displayed in landscape orientation.
“col”	“width”	“n*”	Relative column width expressed as a number followed by an asterisk. A column with width “3*” is 3 times as wide as a column with width “1*”. The exact width of a column depends on the overall width of the table and width of the other columns.
“tr”	“class”	“mapunit”	A table row that begins the data for a map unit.
		“even”	A table row that can be shaded in alternating colors to improve readability. Alternates with “odd”.
		“odd”	A table row that can be shaded in alternating colors to improve readability. Alternates with “even”.
		“interpdata”	A table row containing data about an interpretation in the Survey Area Data Summary report.
		“units”	A table row containing units of measure. It is a type of subheading for the table.
“td”	“rowspan”	a number	Number of rows in the heading occupied by this cell. Part of standard HTML tables.
	“colspan”	a number	Number of columns in the heading occupied by this cell. Part of standard HTML tables.

Element	Attribute Name	Attribute Value	Meaning
	“class”	“begindatagroup”	Cell is the first of a group of cells that are set off visually by a heavier vertical border on the left edge.
		“enddatagroup”	Cell is the last of a group of cells that are set off visually by a heavier vertical border on the right edge.
		“datetime”	Cell contains a date/time field that should be formatted according to the date conventions for the report.
“para”	“role”	“mu-name”	Content is a map unit symbol or name.
		“comp-name”	Content is a component name.
		“number”	Numeric data, normally displayed right justified in a cell.
		“class-name”	Non-numeric data, such as a class name, normally displayed centered in a cell.
		“hang-list”	Multiple values of character type data which are displayed in a vertical list with hanging indents
		“preservenewlines”	Content is a text field that may include newlines, tabs, and significant spaces. Normally this “white space” is removed. This attribute will preserve the layout of the original text.

Examples

The following example creates two view panes side by side.

```
ELEMENT OPEN "div".
ELEMENT  "article" Attrib("style","float:left;margin-
left:0;padding:1em;overflow:hidden").
ELEMENT  "article" Attrib("style","float:right;margin-
left:3;padding:1em;overflow:hidden").
ELEMENT CLOSE "div".
```

The following example creates an area link.

```
ELEMENT "a" ATTRIB("href", "ftp://ftp-
fc.sc.egov.usda.gov/NSSC/GDS/GDS_v4_11.pdf") "GEOMORPHIC
DESCRIPTION SYSTEM".
```

The following example places an image.

```
ELEMENT "img"
ATTRIB("src","https://nrcs.sc.egov.usda.gov/ssra/nssc/Proje
cts/NASIS/triangle.jpg")
ATTRIB ("traget", "_blank") ATTRIB ("alt",""NASIS site link
broken"")
ATTRIB ("width", "600px") ATTRIB ("height", "536px").
```

The following example creates an email link.

```
ELEMENT "a"
ATTRIB("href","mailto:kevin.godsey@mo.usda.gov?Subject=Erro
r%20in%20report") "PLEASE SEND ERRORS BY CLICKING HERE".
```

SVG Scalar Vector Graphics

HTML graphic windows can now be created to display information.

Example

The following example creates a red circle with a black outline with the center located at x=100, y=50 and a radius of 40 pixels.

```
SECTION DATA
ELEMENT OPEN 'html'.
ELEMENT OPEN 'body'.
ELEMENT OPEN 'svg' ATTRIB  ('xmlns',
'http://www.w3.org/2000/svg" version="1.1').
ELEMENT 'circle' ATTRIB  ('cx', '100') ATTRIB  ( 'cy',
'50') ATTRIB ('r', '40') ATTRIB ('stroke',
'black;width;2')ATTRIB ('fill', 'red').
ELEMENT CLOSE 'svg'.
ELEMENT CLOSE 'body'.
ELEMENT CLOSE 'html'.
```

Java Scripts

Java Scripts can now be used in NASIS to perform many tasks in the HTML window. For example, Java Scripts can be used to plot points of sand, silt, and clay on textural triangles; plot point positions on maps; and graph data from NASIS tables.

Each report that includes a java script must contain a reference to the scripting library. Each script needs a metadata section and a contents section. These elements should be in the Header section. They only need to be referenced once.

The following example shows header information that references a scripting library.

```
ELEMENT OPEN 'html'
     ATTRIB ('xmlns','http://www.w3.org/1999/xhtml').
ELEMENT 'meta'
     ATTRIB ('http-equiv', 'content-type')
     ATTRIB ('charset','UTF-8').
ELEMENT  'script'
     ATTRIB  ('src','https://www.google.com/jsapi')
     ATTRIB ('type', 'text/javascript').
```

In the following example, the header is opened, then the Meta data element is defined, then the reference java script library is located, and then a few style elements are defined.

```
ELEMENT OPEN 'head'.
ELEMENT 'meta' ATTRIB ('charset', 'utf-8').
ELEMENT 'script'
     ATTRIB ('type', 'text/javascript') ATTRIB ('src',
     'https://cdnjs.cloudflare.com/ajax/libs/d3/3.4.13/d3.m
     in.js').
ELEMENT 'style'
     ATTRIB ('type', 'text/css') 'body {font: 12px sans-
     serif;}.axis path,.axis line {fill: none;stroke:
     #000;shape-rendering: crispEdges;}.x.axis path
     {display: none;}'.
ELEMENT CLOSE 'head'.
```

The following example opens the d3js library.

```
ELEMENT 'script'
     ATTRIB ('type', 'text/javascript')
     ATTRIB ('src', 'http://d3js.org/d3.v3.min.js').
```

The following example opens a scalable vector graphic (svg) and defines the library and size of the window.

```
ELEMENT open'svg'
     ATTRIB("xmlns", "http://www.w3.org/2000/svg")
     ATTRIB("version","1.1") ATTRIB ("width", "959")
     ATTRIB("height","593").
```

The following example opens two links to the libraries.

```
ELEMENT  'script'
     ATTRIB ('src', 'http://d3js.org/d3.v3.min.js').
ELEMENT  'script'
     ATTRIB ('src', 'http://d3js.org/topojson.v1.min.js').
```

The following example shows the Java Script element that needs to be placed in the data section of the report.

```
ELEMENT OPEN "script"
     ATTRIB ("type", "text/javascript") "script goes here".
ELEMENT CLOSE "script".
```

NASIS can now graph data in many ways within an SVG window. Below is a link to an excellent source for java scripts that create dynamic graphing systems.

https://github.com/d3/d3/wiki/Gallery

The Documentation folder in reports has many examples of charts, graphs, and maps generated from NASIS data using d3 scripts, google API scripts, and basic Java Scripts.

Examples of SQL Date Formats

```
scheduledstartdate  AS date
10/1/2017 12:00:00 AM

Dateadd(YEAR, +1, scheduledstartdate) AS date
10/1/2018 12:00:00 AM

Dateadd(MONTH, -1, scheduledstartdate) AS date
9/1/2017 12:00:00 AM

LEFT(scheduledstartdate, 11) AS date
Oct 1 2017

CONVERT(varchar(11),scheduledstartdate) AS Date
Oct 1 2017

Datename(MONTH,scheduledstartdate)  AS date
October

Datename(YEAR, scheduledstartdate)  AS date
2018

Datename(WEEKDAY, scheduledstartdate) AS date
Sunday
```

```
Datename(QUARTER, scheduledstartdate) AS date
4

datepart(DAY, scheduledstartdate)  AS date
16

DATEDIFF(YEAR, scheduledstartdate, scheduledcompletiondate)
AS date
1

DATEDIFF(YEAR, scheduledstartdate, TODAY) AS Date
1

Find the beginning of the fiscal year.
((YEAR(progrptdate) = fy AND MONTH(progrptdate) < 10)
OR (YEAR(progrptdate) = (fy -1) AND MONTH(progrptdate) >=
10))

assign date dateformat(date, "d").
10/1/2017
```

Appendix II: How to Optimize the SQL Query

Writing Best Practices

- Code the query as simply as possible.
- Use upper-case for all SQL verbs and begin each on a new line.
- Separate all words with a single space.
- Set and maintain a table alias standard; enable aliases to prefix all columns; and when a query involves more than one table, prefix all column names with their aliases.
- Whatever you do, be consistent.
- Pick the best "driving" table and the best join order to eliminate rows as early as possible.
- Write separate SQL statements for specific tasks.

Further Scripting Hints

- Apply math to the constant side of equation >, <, >=, <=, is null, is not null
- Avoid Not in, !=, and Like '%pattern', not exists
- Specify the column names instead of using the asterisk.
    - Do not use asterisk in subqueries. Pick a primary key value, such as *muiid*.
- Make the table with the least number of rows the driving table by making it first in the FROM clause. Use the specific join between the tables with the ON clause instead of using BY DEFAULT.
- A short alias is parsed more quickly than a long table name or alias.
- Use INNER JOINs as much as possible and put the predicates in the ON clause.
- Left joins are only used when data tables are known to be missing.
    - Place predicates (condition statements) in the WHERE clauses if possible.
- Place indexed columns and most limiting conditions first in the predicate clauses.
- Do not select unnecessary columns. Do not include unnecessary GROUP BY or ORDER BY clauses.
- The ORDER BY clause is mandatory in SQL if the sorted result set is expected. The ORDER BY keyword is used to sort the result-set by specified columns. The ORDER BY clause is very costly in terms of computing time.
- Do not use ORDER BY and SORT IN the same query, if possible.
- You cannot use aliases in GROUP BY or ORDER BY.
- Minimize the use of temporary tables. Instead, use table variables and subqueries. In 99% of cases, table variables reside in memory and therefore are a lot faster. Temporary tables reside in the Temp database. Operating on temporary tables requires inter-database communication, and therefore is slower.
- Use EXISTS rather than DISTINCT wherever possible.
- Do not use COUNT to determine whether particular data exists.
- DISTINCT and UNION should be used only if necessary.

    - DISTINCT and UNION operators cause sorting, which slows down the SQL execution.
- Use UNION ALL instead of UNION, if possible. It is much more efficient.
- When doing multiple table joins, consider the benefits and costs for each of EXISTS, IN, and table joins. If the selective predicate is in the subquery, then use IN. If the selective predicate is in the parent query, then use EXISTS.

Symbol operators such as >,<,=,!=, etc. are very helpful.

Examples

Use:

```
SELECT compname FROM component WHERE comppct_r>=10
```

Rather than:

```
SELECT compname FROM component WHERE comppct_r>11
```

Try to avoid the NOT operator in SQL. It is much faster to search for an exact match (positive operator), using LIKE, IN, EXIST, or the = symbol operator than to use a negative operator, such as NOT LIKE, NOT IN, NOT EXIST, or the != symbol.

Use a function instead of a not equals.

Use:

```
SELECT compname FROM component WHERE comppct_r > 0;
```

Rather than:

```
SELECT compname FROM component WHERE comppct_r !=0;
```

Use BETWEEN instead of >=.

Use a non-column expression on the left side of the operator because it will be processed earlier and put math functions on the right side of the operator.

Use:

```
WHERE comppct_r < 50/100);
```

Rather than:

```
WHERE comppct_r *100 < 50;
```

Limit the number of results by using the TOP() function in the SELECT clause. A table can have a few million records, and a search query without a limitation will just slow it down.

Minimize the number of subquery blocks in your query. Note, however, that several small queries can run faster than one big one if there are a lot of joins.

If the data is in only one table, it is still faster to join to another table to create indexing.

Use:

```
SELECT muname
FROM mapunit
INNER join correlation on
correlation.muiidref=mapunit.muiid AND muname=”Menfro”;
```

Rather than:

```
SELECT muname
FROM mapunit
WHERE muname = “Menfro”;
```

Ensure repeated SQL statements are written *absolutely identically* to facilitate efficient reuse: re-parsing can often be avoided for each subsequent use.

Avoid a HAVING clause in SELECT statements. It only filters selected rows after all the rows have been returned. Use HAVING only when summary operations applied to columns are restricted by the clause. A WHERE clause may be more efficient.

Use:

```
SELECT compname FROM component WHERE compname!= 'Vancouver'
AND compname!= 'Toronto'; GROUP BY compname;
```

Rather than:

```
SELECT compname FROM component GROUP BY compname HAVING
compname!= 'Vancouver' AND compname!= 'Toronto';
```

The UNION operator is faster than using OR.

The UNION statement allows you to combine the result sets of 2 or more select queries.

Try to use UNION ALL in place of UNION. UNION eliminates duplicates and takes processing time to complete.

Use:

```
SELECT cropname FROM componentcrop WHERE cropname=”corn”
UNION ALL
SELECT cropname FROM componentcrop WHERE cropname =”beans”;
```

Rather than:

```
SELECT cropname
FROM componentcrop
WHERE cropname=”corn” or cropname =”beans”
```

Appendix III: Expanded SQL Capabilities in NASIS

Programmers have added new data manipulation functions in the Newer SQL Server, which will become available in NASIS in a future release. Some of these functions provide powerful data aggregation capabilities that could reduce the need to use the AGGREGATE feature in NASIS reports. These functions could substantially reduce the memory requirements for running reports and thereby allow bigger reports to be run.

Complete documentation of SQL Server is at the Microsoft site "Transact-SQL Reference" at http://technet.microsoft.com/en-us/library/bb510741.aspx.

Many of the new features are under the heading "Built-in Functions." Some of them are complex. Careful reading and experimenting may be needed to understand how they work.

Following is a brief report script that illustrates several of the new functions.

Script	Comments
`EXEC SQL SELECT compname, majcompflag, slope_r`	Aggregates functions with partitioning by component name. Doesn't require GROUP BY. Is more efficient than a subquery.

`,AVG(slope_r) over(partition by compname) as avg_slope`
`,MIN(slope_r) over(partition by compname) as min_slope`
`,MAX(slope_r) over(partition by compname) as max_slope`
	Applies analytical functions with partitioning by component name.

`,rank() over(partition by compname, majcompflag order by slope_r) as slope_rank`	Gets ranking of slope values in partition.
`,lag(slope_r) over(partition by compname, majcompflag order by slope_r) as prev_slope`	Gets slope value from previous record in partition.

`,last_value(tfact) over(partition by compname order by slope_r`
 	
`rows between unbounded preceding and unbounded following) as last_tfact`
	Gets value of tfact from component with the largest slope in partition.

,percentile_cont (0.75) within group (order by slope_r) over(partition by compname) as slope_75pctile	Gets the 75th percentile slope value by partition.
,datepart(weekday, recwlupdated) day ,datename(month, dateadd(month, 1, current_timestamp)) as nextmo ,datediff(day, recwlupdated, current_timestamp) as age	Applies functions for manipulating dates.
FROM component WHERE slope_r is not null order by compname, slope_r;.	

Aggregate Functions with OVER clause

The OVER clause can be used with an aggregate function in a manner that is similar to, but more efficient than, a subquery. For example, the aggregate function

```
AVG(slope_r) over(partition by compname)
```

produces the same results as the subquery

```
(SELECT AVG(slope_r)FROM component c1 WHERE
c1.compname=component.compname)
```

A significant difference is that partitioning requires that the query uses ORDER BY for the partitioning column(s) in a manner similar to the way AGGREGATE works in NASIS.

The OVER clause is also similar to GROUP BY but is less restrictive. GROUP BY requires everything in the SELECT clause must be either in the GROUP BY list or an aggregate function. An aggregate function with OVER does not require GROUP BY.

All of the standard SQL aggregate functions are now allowed in NASIS and can be used with OVER. They are: COUNT, COUNT_BIG, AVG, CHECKSUM_AGG, MAX, MIN, SUM, STDEV, STDEVP, VAR, and VARP.

Ranking Functions

Ranking functions create a ranking value for each row in a query. They allow the query results to be divided into sorted partitions before computing the ranking value, but partitioning is not required. If the ranking is not partitioned, the results are computed over the whole set of data

returned by the query; in other words, there is just one partition. Ranking functions always require an ORDER BY within the OVER clause.

Four ranking functions are available. Based on the example in the report script above, the four functions would be:

```
,rank() over(partition by compname, majcompflag
  order by slope_r) as slope_rank
,dense_rank() over(partition by compname, majcompflag
  order by slope_r) as slope_dense_rank
,row_number() over(partition by compname, majcompflag
  order by slope_r) as slope_row_number
,ntile(4) over(partition by compname, majcompflag
  order by slope_r) as slope_quartile
```

These functions say that within each group of rows that has the same values for compname and majcompflag, a ranking is computed based on slope_r. The NTILE function requires a number to specify how many subgroups are used. For example, NTILE(4) means to compute the quartile for each row. The following sample of the output is shaded to show how the rows are partitioned.

compname	majcompflag	slope_r	slope_ rank	slope_dense_ rank	slope_row_ number	slope_ quartile
Aquults	0	1.0	1	1	1	1
Aquults	0	1.0	1	1	2	2
Avilla	1	2.0	1	1	1	1
Avilla	1	4.0	2	2	2	2
Avilla	1	5.0	3	3	3	3
Avilla	1	9.0	4	4	4	4
Bengal	0	12.0	1	1	1	1
Bengal	1	25.0	1	1	1	1
Bengal	1	47.5	2	2	2	2
Bismarck	1	5.0	1	1	1	1
Bismarck	1	5.0	1	1	2	1
Bismarck	1	5.5	3	2	3	2
Bismarck	1	12.0	4	3	4	2
Bismarck	1	12.0	4	3	5	3
Bismarck	1	12.0	4	3	6	3
Bismarck	1	25.0	7	4	7	4
Bismarck	1	52.5	8	5	8	4

Date and Time Functions

Several operations can be performed in SQL on columns that contain date and time data. In NASIS, the value of a date/time column is always processed as a character string, which means

the column is difficult to manipulate and format. By using these SQL functions in the query, it is typically not be necessary to further manipulate the data with NASIS DEFINE statements.

- CURRENT_TIMESTAMP and GETDATE() return the current date and time (to the millisecond) on the server. The values are based on Central time for reports run on the Kansas City servers and on local time for reports run on the workstation.
- GETUTCDATE() returns the current date and time as UTC time (a.k.a., Greenwich time zone).
- DATENAME(*datepart, date*) returns the character typed value of a specified part of a date value. The date parameter could be something like GETDATE() or the name of a database column that contains a date value, such as "progrptdate".
- The datepart is one of the words YEAR, QUARTER, MONTH, DAY, DAYOFYEAR, WEEK, WEEKDAY, HOUR, MINUTE, SECOND, or TZOFFSET. For example, `DATENAME(MONTH, progrptdate)` might return "March" as the month when the progress item was reported.
- DATEPART(*datepart, date*) is like DATENAME, but it returns the numeric value of the specified part of the date. Using the previous example, `DATEPART(MONTH, progrptdate)` would return 3. For some date parts, there is no equivalent word; and, therefore, DATENAME and DATEPART return the same value. DAY is always the number of the day, but WEEKDAY could be "Monday" or 2.
- YEAR(*date*), MONTH(*date*), and DAY(*date*) are the same as DATEPART(*YEAR, date*), DATEPART(*MONTH, date*), and DATEPART(*DAY, date*), respectively.
- DATEADD(*datepart, number, date*) produces a date/time value by adding a specified amount to a part of the date. For example `DATEADD(YEAR, -1, progrptdate)` subtracts one year from the value of "progrptdate" and `DATEADD(WEEK, 2, progrptdate)` adds 2 weeks to the date.
- DATEDIFF(*datepart, startdate, enddate*) computes the amount of the specified date part that elapsed between the starting and ending dates. For example, `DATEDIFF(MONTH, progrptdate, GETDATE())` gives the number of months since the progress item was reported.
- Other functions listed in the SQL reference manual are also usable in NASIS in most cases.

Other Analytic Functions

SQL Server provides additional functions similar to the ranking functions. More details are in the SQL reference at http://technet.microsoft.com/en-us/library/bb510741.aspx. The following functions are supported by NASIS.

- FIRST_VALUE and LAST_VALUE provide the value of a specified column from the first or last row of a partition (similarly to the FIRST and LAST aggregations in NASIS).
- LAG and LEAD provide the value of a specified column from the n^{th} preceding or following row in the partition (similarly to the functionality of ARRAYSHIFT in NASIS).
- CUM_DIST and PERCENT_RANK give measures of the relative position of a row within the partition.
- PERCENTILE_CONT and PERCENTILE_DISC compute the value of the n^{th} percentile for the values in the partition.

When using these functions, you should note that the default method of calculation uses the rows starting at the beginning of the partition up to the current row, which may not be what you expect. By default, the LAST_VALUE function provides the value in the current row, rather than the last row in the partition as you might expect. The report script example above shows how to declare a set of rows for the calculation that is different from the default. In the following case, the LAST_ROW calculation uses all the rows in the partition.

```
,last_value(tfact) over(partition by compname order by
slope_r  rows between unbounded preceding and unbounded
following) as last_tfact
```

Appendix IV: Common Error Messages

The following error messages appear for the listed conditions.

When text is identified on the general tab and html format on the report tab:

```
Preparing to run report "HTML-soil extent map" on the local database...
ERROR
While running ReportScript "HTML-soil extent map"
ELEMENT cannot be used in a report whose output format is TXT
```

When html is checked on the general tab and the data is text:

```
ERROR
While running ReportScript "UTIL - Topsoiling"
  with coiid=1195676
A USING statement has more columns than the TEMPLATE "tier1"
```

When page width and length are not unlimited and output is text:

```
ERROR
While running ReportScript "test errrors"
Count must be positive and count must refer to a location within the string/array/collection.
Parameter name: count
```

When the second column is not aggregated to none (REGROUP error):

```
ERROR
While running ReportScript "test errrors"
Error in DEFINE grpgrp
Variables used in REGROUP have unequal numbers of values (2 vs. 1)
```

When a period is missing, the error references the first column of the line below the line that is missing the period:

```
Preparing to run report "test errrors" on the local database...
While verifying ReportScript "test errrors"
Syntax error at line 16, column 1
'page' is out of context.
```

When a Right parenthesis is missing:

```
Preparing to run report "test errrors" on the local database...
While verifying ReportScript "test errrors"
Error in query beginning at line 3
expecting "RPAREN", found 'FROM'
```

When the left parenthesis is missing:

```
Preparing to run report "test errrors" on the local database...
While verifying ReportScript "test errrors"
Syntax error at line 4, column 28
')' is out of context.
```

When you codelabel a column but forget to give it an alias:

```
Preparing to run report "test errrors" on the local database...
While verifying ReportScript "test errrors"
Error in query beginning at line 3
An expression in the Select clause must have an alias
```

When you have an extra comma at the end of the select list:

```
Preparing to run report "test errrors" on the local database...
While verifying ReportScript "test errrors"
Syntax error at line 5, column 1
'FROM' is out of context.
```

If you forget to open the HTML and body:

```
Preparing to run report "test report" on the local database...
ERROR
While running ReportScript "test report"
There are multiple root elements. Line 2, position 2.
```

When you try to concatenate the areasymbol and the musym:

```
Preparing to run report "test concatenate" on the local database...
ERROR
While running ReportScript "test concatenate"
Unable to run the query that starts at line 1
Implicit conversion of varchar value to varchar cannot be performed because the collation of the value is unresolved due to a collation conflict between "Latin1_General_BIN" and "SQL_Latin1_General_Pref_CP1_CI_AS" in add operator.
```

Appendix V: HTML Formatting

The following table provides a description of standard HTML tags.

Tag	Description
<!--...-->	Defines a comment
<!DOCTYPE>	Defines the document type
<a>	Defines an anchor
<abbr>	Defines an abbreviation
<acronym>	Defines an acronym
<address>	Defines contact information for the author/owner of a document
<area />	Defines an area inside an image-map
<b>	Defines bold text
<base />	Defines a default address or a default target for all links on a page
<bdo>	Defines the text direction
<big>	Defines big text
<blockquote>	Defines a long quotation
<body>	Defines the document's body
 	Defines a single line break
<button>	Defines a push button
<caption>	Defines a table caption
<cite>	Defines a citation
<code>	Defines computer code text
<col />	Defines attribute values for one or more columns in a table
<colgroup>	Defines a group of columns in a table for formatting
<dd>	Defines a description of a term in a definition list
<del>	Defines deleted text
<dfn>	Defines a definition term
	Defines a section in a document
<dl>	Defines a definition list
<dt>	Defines a term (an item) in a definition list
	Defines emphasized text
<fieldset>	Defines a border around elements in a form
<form>	Defines an HTML form for user input
<frame />	Defines a window (a frame) in a frameset
<frameset>	Defines a set of frames
<h1> to <h6>	Defines HTML headings
<head>	Defines information about the document
<hr />	Defines a horizontal line
<html>	Defines an HTML document

Tag	Description
<i>	Defines italic text
<iframe>	Defines an inline frame
<img />	Defines an image
<input />	Defines an input control
<ins>	Defines inserted text
<kbd>	Defines keyboard text
<label>	Defines a label for an input element
<legend>	Defines a caption for a fieldset element
	Defines a list item
<link />	Defines the relationship between a document and an external resource
<map>	Defines an image-map
<meta />	Defines metadata about an HTML document
<noframes>	Defines an alternate content for users that do not support frames
<noscript>	Defines an alternate content for users that do not support client-side scripts
<object>	Defines an embedded object
	Defines an ordered list
<optgroup>	Defines a group of related options in a select list
<option>	Defines an option in a select list
<p>	Defines a paragraph
<param />	Defines a parameter for an object
<pre>	Defines preformatted text
<q>	Defines a short quotation
<samp>	Defines sample computer code
<script>	Defines a client-side script
<select>	Defines a select list (drop-down list)
<small>	Defines small text
	Defines a section in a document
	Defines strong text
<style>	Defines style information for a document
<sub>	Defines subscripted text
<sup>	Defines superscripted text
<table>	Defines a table
<tbody>	Groups the body content in a table
<td>	Defines a cell in a table
<textarea>	Defines a multi-line text input control
<tfoot>	Groups the footer content in a table
<th>	Defines a header cell in a table
<thead>	Groups the header content in a table

Tag	Description
<title>	Defines the title of a document
<tr>	Defines a row in a table
<tt>	Defines teletype text
	Defines an unordered list
<var>	Defines a variable part of a text

Appendix VI: Default HTML Output Format

The following shows the values from the cascading style sheet used for HTML output from NASIS.

```
{font-family:Verdana, Arial, Helv, Sans-Serif;font-size: 10pt;}

Soilprop = {FONT: 10pt Verdana, Helvetica, Arial, sans-serif;
COLOR: #000000;TEXT-ALIGN: left; LINE-HEIGHT: 12pt; MARGIN-LEFT:
12pt;TEXT-INDENT: -12pt;  MARGIN-TOP: 0pt;MARGIN-BOTTOM: 0pt;}

div.SoilReport h1 = {font-size: 16px;}

div.SoilReport h2= {font-size: 14px;}

div.SoilReport h3, - h6 = {font-size: 12px;}

div.menu table td = {font-size: 11px;}

ul.noimage li = {list-style-type: none; list-style-image: none;}

ul.noimage li p = {margin-left: 4ex;}

ul.noimage li h1-h6 = {margin-bottom: 0px;}

first = {margin-top: 0px !important;}

numeric.number = {text-align: right;}

bold = {font-weight: bold; margin-right: 1ex;}

super = {vertical-align: super; font-size: smaller;}

sub = {vertical-align: sub; font-size: smaller;}

padded = {padding: 5px !important;}

unpadded = {padding: 0px !important;}

scrollable = {overflow-y: auto !important;}

nowrap = {white-space: nowrap;}

Table = {border-collapse: collapse;}

table.data th-td = {border-width: 1px; border-style: solid;
padding: 4px;}
```

```
table.data td.begindatagroup = {border-left-width: 2px;border-
left-style: solid;}

table.data td.enddatagroup = {border-right-width: 2px;border-
right-style: solid;}

table.data td.label = {text-align: right; padding: 5px;}

table.data th = {width: 100%; padding: 2px !important;}

table.data th.title div.title = {font-weight: bold; padding-
left: 1.25em;text-indent: -1em;text-align: left;}

table.data th.control = {vertical-align: middle; text-align:
right;}

table.data td = {text-align: left; vertical-align: top;}

table.data td.columnhead, tr.columnhead td = {padding: 2px;font-
weight: bold; text-align: center;}

table.data tr.units td = {font-style: italic; text-align:
center;}

table.data td p.mu-name = {padding-left: 0.5em; text-indent: -
0.5em;}

table.data td p.comp-name = {padding-left: 1em;text-indent: -
0.5em;}

table.data td p.class-name = {text-align: center;}

table.data td p.reason0 = {text-align: left; padding-left:
0.5em;text-indent: -0.5em;}

table.data td p.reason1 = {text-align: left; padding-left:
1em;text-indent: -0.5em;}

table.data td p.reason2 = {text-align: left; padding-left:
1.5em;text-indent: -0.5em;}

table.data td p.hang-list = {text-align: left; padding-left:
0.5em;text-indent: -0.5em;vertical-align: middle;}

table.data tfoot td = {font-weight: bold; padding-top: 5px;}

table.data tr.even td = {background-color: #FDFDEE;}
```

```
table.data tr.odd td = {background-color: #F8F8D8;}

div.menu table = {color: #000000; background-color: #FBFBE8;}

div.menu table td = {border-color: #000000;}

div.menu table td.over = {background-color: #E5DEBC !important;}

title.muname = {FONT: 14pt bold Helvetica, Arial, sans-serif;
COLOR: #000000; TEXT-ALIGN: left; LINE-HEIGHT: 16pt; MARGIN-TOP:
18pt; MARGIN-BOTTOM: 3pt;}

subtitle = {FONT: 12pt bold Helvetica, Arial, sans-serif; COLOR:
#000000; TEXT-ALIGN: left;
LINE-HEIGHT: 14pt; MARGIN-TOP: 6pt; MARGIN-BOTTOM: 3pt;}

headnote = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; TEXT-ALIGN: center;
LINE-HEIGHT: 10pt; MARGIN: 6pt 8pt 6pt 8pt;}

footnote = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; TEXT-ALIGN: left;
 LINE-HEIGHT: 10pt; MARGIN: 6pt 0pt 0pt 0pt;}

reportdesc = {FONT: 10pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 12pt;
    TEXT-ALIGN: left; TEXT-INDENT: 12pt; MARGIN-TOP: 0pt;
MARGIN-BOTTOM: 0pt;}

P = {FONT: 10pt Helvetica, Arial, sans-serif; COLOR: #000000;
TEXT-ALIGN: left; LINE-HEIGHT: 12pt;
 MARGIN-TOP: 0pt; MARGIN-BOTTOM: 0pt;}

TD = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR: #000000;
LINE-HEIGHT: 10pt; MARGIN: 0pt 0pt 0pt 0pt; PADDING: 0pt 2pt 0pt
2pt;}

THEAD TR TD = {FONT: bold 8pt Helvetica, Arial, sans-serif;
COLOR: #000000; LINE-HEIGHT: 10pt; TEXT-ALIGN: center; MARGIN:
0pt 0pt 0pt 0pt; PADDING: 0pt 0pt 0pt 0pt; BACKGROUND-COLOR:
#E8E8E8;}

mu-name = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt; TEXT-ALIGN: left; MARGIN-LEFT: 6pt;
TEXT-INDENT: -6pt; MARGIN-TOP: 0pt; MARGIN-BOTTOM: 0pt;}
```

```
comp-name = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt; TEXT-ALIGN: left; MARGIN-LEFT: 12pt;
TEXT-INDENT: -6pt; MARGIN-TOP: 0pt; MARGIN-BOTTOM: 0pt;}

units = {FONT: italic 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt; TEXT-ALIGN: center; MARGIN-LEFT:
0pt; TEXT-INDENT: 0pt; MARGIN-TOP: 0pt; MARGIN-BOTTOM: pt;}

class-name = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt; TEXT-ALIGN: center; MARGIN-LEFT:
0pt; TEXT-INDENT: 0pt; MARGIN-TOP: 0pt; MARGIN-BOTTOM: 0pt;}

hang-list = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt; TEXT-ALIGN: left; MARGIN-LEFT: 6pt;
TEXT-INDENT: -6pt; MARGIN-TOP: 0pt; MARGIN-BOTTOM: 0pt;}

reason0 = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt; TEXT-ALIGN: left; MARGIN-LEFT: 6pt;
TEXT-INDENT: -6pt; MARGIN-TOP: 0pt; MARGIN-BOTTOM: 0pt;}

reason1 = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt; TEXT-ALIGN: left;
MARGIN-LEFT: 12pt; TEXT-INDENT: -6pt; MARGIN-TOP: 0pt; MARGIN-
BOTTOM: 0pt;}

reason2 = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt;
TEXT-ALIGN: left; MARGIN-LEFT: 18pt; TEXT-INDENT: -6pt; MARGIN-
TOP: 0pt; MARGIN-BOTTOM: 0pt;}

reason3 = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt;
TEXT-ALIGN: left; MARGIN-LEFT:24pt; TEXT-INDENT: -6pt; MARGIN-
TOP: 0pt; MARGIN-BOTTOM: 0pt;}

Rating value = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt; TEXT-ALIGN: right; MARGIN-LEFT: 0pt;
MARGIN-TOP: 0pt; MARGIN-BOTTOM: 0pt;}

number = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt;
TEXT-ALIGN: right; MARGIN-LEFT: 0pt; MARGIN-TOP: 0pt; MARGIN-
BOTTOM: 0pt;}

total-line = {FONT: 8pt Helvetica, Arial, sans-serif; COLOR:
#000000; LINE-HEIGHT: 10pt;
PADDING: 6pt 2pt 0pt 2pt; BORDER-TOP: 1px solid black;}
```

```
shade0 = {background-color: #FFEEDD;}

shade1 = {background-color: white;}

descrip-mu = {FONT: 10pt Helvetica, Arial, sans-serif; COLOR:
#000000; TEXT-ALIGN: left; LINE-HEIGHT: 12pt; MARGIN-LEFT: 0pt;
MARGIN-TOP: 12pt; MARGIN-BOTTOM: 0pt;}

descrip-text = {FONT: 10pt Helvetica, Arial, sans-serif; COLOR:
#000000; TEXT-ALIGN: left; LINE-HEIGHT: 12pt; MARGIN-LEFT: 18pt;
MARGIN-TOP: 12pt; MARGIN-BOTTOM: 0pt;}
```

Appendix VII: Color Coding

The following shows the color values used in HTML in RGB and Hex.

Color Name	Col	R	G	B	Hex
lightpink		255	182	193	#FFB6C1
pink		255	192	203	#FFC0CB
crimson		220	20	60	#DC143C
lavenderblush		255	240	245	#FFF0F5
palevioletred		219	112	147	#DB7093
hotpink		255	105	180	#FF69B4
deeppink		255	20	147	#FF1493
mediumvioletred		199	21	133	#C71585
orchid		218	112	214	#DA70D6
thistle		216	191	216	#D8BFD8
plum		221	160	221	#DDA0DD
violet		238	130	238	#EE82EE
fuchsia*		255	0	255	#FF00FF
fuchsia*		255	0	255	#FF00FF
darkmagenta		139	0	139	#8B008B
purple*		128	0	128	#800080
mediumorchid		186	85	211	#BA55D3
darkviolet		148	0	211	#9400D3
darkorchid		153	50	204	#9932CC
indigo		75	0	130	#4B0082
blueviolet		138	43	226	#8A2BE2
mediumpurple		147	112	219	#9370DB
mediumslateblue		123	104	238	#7B68EE
slateblue		106	90	205	#6A5ACD
darkslateblue		72	61	139	#483D8B
ghostwhite		248	248	255	#F8F8FF
lavender		230	230	250	#E6E6FA
blue*		0	0	255	#0000FF
mediumblue		0	0	205	#0000CD
darkblue		0	0	139	#00008B
navy*		0	0	128	#000080
midnightblue		25	25	112	#191970
royalblue		65	105	225	#4169E1
cornflowerblue		100	149	237	#6495ED
lightsteelblue		176	196	222	#B0C4DE
lightslategray		119	136	153	#778899
slategray		112	128	144	#708090
dodgerblue		30	144	255	#1E90FF
aliceblue		240	248	255	#F0F8FF
steelblue		70	130	180	#4682B4

Color Name	Col	R	G	B	Hex
lightskyblue		135	206	250	#87CEFA
skyblue		135	206	235	#87CEEB
deepskyblue		0	191	255	#00BFFF
lightblue		173	216	230	#ADD8E6
powderblue		176	224	230	#B0E0E6
cadetblue		95	158	160	#5F9EA0
darkturquoise		0	206	209	#00CED1
azure		240	255	255	#F0FFFF
lightcyan		224	255	255	#E0FFFF
paleturquoise		175	238	238	#AFEEEE
aqua*		0	255	255	#00FFFF
aqua*		0	255	255	#00FFFF
darkcyan		0	139	139	#008B8B
teal*		0	128	128	#008080
darkslategray		47	79	79	#2F4F4F
mediumturquoise		72	209	204	#48D1CC
lightseagreen		32	178	170	#20B2AA
turquoise		64	224	208	#40E0D0
aquamarine		127	255	212	#7FFFD4
mediumaquamarine		102	205	170	#66CDAA
mediumspringgreen		0	250	154	#00FA9A
mintcream		245	255	250	#F5FFFA
springgreen		0	255	127	#00FF7F
mediumseagreen		60	179	113	#3CB371
seagreen		46	139	87	#2E8B57
honeydew		240	255	240	#F0FFF0
darkseagreen		143	188	143	#8FBC8F
palegreen		152	251	152	#98FB98
lightgreen		144	238	144	#90EE90
limegreen		50	205	50	#32CD32
lime*		0	255	0	#00FF00
forestgreen		34	139	34	#228B22
green*		0	128	0	#008000
darkgreen		0	100	0	#006400
lawngreen		124	252	0	#7CFC00
chartreuse		127	255	0	#7FFF00
greenyellow		173	255	47	#ADFF2F
darkolivegreen		85	107	47	#556B2F
yellowgreen		154	205	50	#9ACD32

Color Name	Col	R	G	B	Hex
olivedrab		107	142	35	#6B8E23
ivory		255	255	240	#FFFFF0
beige		245	245	220	#F5F5DC
lightyellow		255	255	224	#FFFFE0
lightgoldenrodyellow		250	250	210	#FAFAD2
yellow*		255	255	0	#FFFF00
olive*		128	128	0	#808000
darkkhaki		189	183	107	#BDB76B
palegoldenrod		238	232	170	#EEE8AA
lemonchiffon		255	250	205	#FFFACD
khaki		240	230	140	#F0E68C
gold		255	215	0	#FFD700
cornsilk		255	248	220	#FFF8DC
goldenrod		218	165	32	#DAA520
darkgoldenrod		184	134	11	#B8860B
floralwhite		255	250	240	#FFFAF0
oldlace		253	245	230	#FDF5E6
wheat		245	222	179	#F5DEB3
orange*		255	165	0	#FFA500
moccasin		255	228	181	#FFE4B5
papayawhip		255	239	213	#FFEFD5
blanchedalmond		255	235	205	#FFEBCD
navajowhite		255	222	173	#FFDEAD
antiquewhite		250	235	215	#FAEBD7
tan		210	180	140	#D2B48C
burlywood		222	184	135	#DEB887
darkorange		255	140	0	#FF8C00
bisque		255	228	196	#FFE4C4
linen		250	240	230	#FAF0E6
peru		205	133	63	#CD853F
peachpuff		255	218	185	#FFDAB9

Color Name	Col	R	G	B	Hex
sandybrown		244	164	96	#F4A460
chocolate		210	105	30	#D2691E
saddlebrown		139	69	19	#8B4513
seashell		255	245	238	#FFF5EE
sienna		160	82	45	#A0522D
lightsalmon		255	160	122	#FFA07A
coral		255	127	80	#FF7F50
orangered		255	69	0	#FF4500
darksalmon		233	150	122	#E9967A
tomato		255	99	71	#FF6347
salmon		250	128	114	#FA8072
mistyrose		255	228	225	#FFE4E1
lightcoral		240	128	128	#F08080
snow		255	250	250	#FFFAFA
rosybrown		188	143	143	#BC8F8F
indianred		205	92	92	#CD5C5C
red*		255	0	0	#FF0000
brown		165	42	42	#A52A2A
firebrick		178	34	34	#B22222
darkred		139	0	0	#8B0000
maroon*		128	0	0	#800000
		255	255	255	#FFFFFF
whitesmoke		245	245	245	#F5F5F5
gainsboro		220	220	220	#DCDCDC
lightgrey		211	211	211	#D3D3D3
silver*		192	192	192	#C0C0C0
darkgray		169	169	169	#A9A9A9
gray*		128	128	128	#808080
dimgray		105	105	105	#696969
black*		0	0	0	#000000