

TQL Editor User Guide

Version 4.9

Introduction

The DTS Terminology Query Language (TQL) is a language for expressing statements (queries) that manipulate or export information found in the DTS Knowledgebase. TQL provides an alternative to direct API programming for more easily performing many common DTS maintenance, Q/A, and output tasks. The TQL Editor, available as both a DTS plug-in and standalone application, permits the creation, editing, saving and interactive execution of TQL queries. A TQL Class is also available to permit execution of queries from batch files and Java applications.

This **TQL Editor User Guide** provides a comprehensive description of the TQL Language and the operation of the TQL Editor. The **TQL Reference Guide**, also in the distribution kit, is a quick reference for TQL syntax formatted as a concise double-sided “trifold”. For previous users of TQL, the **TQL V4.9 Release Notes** are recommended as a brief update on new features.

The TQL Commander is a companion application for running parameterized TQL queries. See the **Parameterized Queries** section below for information on creating of parameterized queries. For instructions on using TQL Commander, see the **TQL Commander User Guide**.

Installation

Extract the files in TQLEditor-4.9.zip into your *DTSInstall* directory. Be sure the Use folder names box is checked. This will place all the TQL Editor files into the appropriate folders:

<u>Folder</u>	<u>Files</u>
<i>DTSInstall</i> \bin\tqleditor	TQLEditor.bat TQLCommander.bat TQL.bat thesaurus-schema-v5.xsd extension-schema-v5.xsd ontylog-schema-V5.xsd subset-schema-v4.xsd TQL 4.5 Javadoc.zip
<i>DTSInstall</i> \lib\modules	tqleditor.jar pluginutils.jar xmldigester.jar poi-3.6-20091214.jar poi-ooxml-3.6-20091214.jar poi-ooxml-schemas-3.6-20091214.jar
<i>DTSInstall</i> \docs	tqleditoruserguide.pdf tqlcommanderuserguide.pdf TQL V4.9 Release Notes.pdf TQL V4.9 Reference Guide.pdf
<i>DTSInstall</i> \docs\help	tqleditoruserguide.htm tqlcommanderuserguide.htm

As desired, edit `TQLEditor.bat`, which runs the TQL Editor as a standalone application, `TQL.bat`, which runs TQL specification files in batch mode, and `TQLCommander.bat`, which runs the TQL Commander application, to accommodate your particular DTS environment.

Operation

TQL queries are sequences of English-like statements that permit the display, modification and export of DTS Knowledgebase information. TQL applications support the creation and/or execution of TQL queries. The primary TQL applications are:

- The TQL Editor, used to interactively create, maintain, and execute TQL queries. The TQL Editor is available as both a DTS Editor Module and a standalone application (accessed by running `TQLEditor.bat` from the `DTSInstall\bin\tqleditor` folder). The TQL Editor (and the TQL Commander) can be invoked via icons on the toolbar in the standard DTS Editor layout and as floating panels by selecting `Tools|New TQL Editor` (and `Tools|New TQL Commander`) from the menu bar. TQL implements the DTS V4 Module Architecture and can be used in custom DTS Editor layouts.
- The TQL Class, used to execute TQL query strings from either batch files or Java applications via the TQL API. For further information, see *Using the TQL Class* later in this Guide.

The following sections describe TQL and discuss how to create and run a TQL query using the TQL Editor.

The Terminology Query Language

Overview

A TQL *query* is a sequence of structured *statements* that describe a desired action on DTS terminology objects. There are thirteen types of *statements* and each statement type may have multiple forms. The *statements* are:

- The *collection-statement* which collects a set of Concepts or Terms for use by subsequent (subordinate) *statements*.
- The *conditional-statement* which supports conditional execution of *statements* based on the result of a boolean expression. The *conditional-statement* implements the familiar IF/ELSE construct used in other programming languages.
- The *create-context-statement* which creates *contexts*, or named groups, of Concepts. Statement forms are available to create DTS Namespaces, Subsets and Authorities. Additional forms create the TQL-specific contexts ConSets and TermSets: local files that can reference sets of Concepts and Terms.
- The *delete-context-statement* which deletes DTS and TQL *contexts*: Namespaces, Subsets, Authorities, ConSets and TermSets.
- The *rename-context-statement* which renames DTS Namespaces, Subsets, and Authorities.
- The *edit-statement* which modifies existing DTS Knowledgebase information. Forms are available to create and delete Concepts and Terms, and add, update and delete Concept and Term Attributes.
- The *export-statement* which creates an export file (or list) of Concept and Term Attribute values. Delimited text (.txt), Excel (.xls and .xlsx), and XML (.xml) output formats are supported. Some forms of the *export-statement* export the complete contents of a Namespace, or definition of a Subset, to XML files. These files can be used for backup or exchange purposes.
- The *for-statement* which allows iteration of its scope over instances of a DTS attribute.
- The *output-statement* which adds messages to the log or export data streams.
- The *set-variables-statement* which sets the values of internal TQL variables.
- The *read-statement* which allows the operations in a *query* to be directed by values in an external file.
- The *constrain-statement* and *parameter-statement* that define the use of parameters in TQL queries.

In the sections that follow, TQL statement types, forms, and element names are shown in *italics*, and element keywords are shown in CAPITALS. Brackets enclose optional TQL elements, vertical bars denote alternation (alternate element forms), and literal characters are enclosed in quote marks. The brackets, vertical bars, and quotes are **not** part of the TQL statement. When writing a query, keywords may be entered in any case and any amount of *whitespace* (including spaces, tabs and returns) can be placed between TQL statements, keywords and elements to improve readability. Either single or double quotes can be used to enclose string literals in *statements*, and within a literal, the backslash character, “\” can be used to escape both quote marks (“\” and “\’”), the tab character (“\t”) and itself (“\\”). TQL also supports escaping the caret (“^”) so that this character can appear in Namespace, Subset and Attribute names. Finally, note that every TQL statement must end in a semicolon.

The TQL is formally described via the simplified BNF grammar given in Appendix A. As an example of this BNF, the “production rule” (definition) of a TQL *query* is:

query := *statement* [*query*]

This is read as “a *query* is a *statement*, optionally followed by another *query*”, i.e., a *query* is a sequence of any number of *statements*.

Similarly, a *statement* is defined as one of the statement types (using the vertical bar to denote alternation):

<i>statement</i> :=	<i>collection-statement</i>	
	<i>conditional-statement</i>	
	<i>create-context-statement</i>	
	<i>delete-context-statement</i>	
	<i>rename-context-statement</i>	
	<i>edit-statement</i>	
	<i>export-statement</i>	
	<i>for-statement</i>	
	<i>output-statement</i>	
	<i>set-variables-statement</i>	
	<i>read-statement</i>	
	<i>parameter-statement</i>	
	<i>constrain-statement</i>	

“A *statement* is a *collection-statement*, or a *conditional-statement*, or ...”

Statement Classifications

Certain *statements* may only be used in specific locations in a *query* or in conjunction with other *statements*. An unrestricted *statement* can appear anywhere in a *query*, including within the scope of another *statement*. A contextual *statement* is dependent on a *context*; it must appear in the scope of a *collection-statement* (see **The Collection Statement** section below for details). Contextual statements, in turn, can be unrestricted or restricted. An unrestricted contextual *statement* can occur anywhere in the scope of its enclosing *collection-statement* and appear at any nesting level (see **The Collection Statement** and **The Conditional Statement** sections below for details on nested *statements*). Multiple unrestricted contextual *statements* can be present in the scope. Similarly, restricted contextual *statements* can occur anywhere in the scope and appear at any nesting level, but there can be ONLY ONE restricted contextual *statement* in the scope and NO unrestricted contextual *statements* may appear in the scope. Finally, a non-contextual *statement* can appear anywhere EXCEPT in the scope of a *collection-statement*. While there are exceptions, non-contextual *statements* usually only appear at the top level of a *query*, i.e., not within the scope of another *statement*.

Some *statements* can be used in multiple positions, often dependent on the *statements*’ arguments. See **Table 1** below and the individual *statement* sections for further information.

Table 1 – TQL Statement Classifications

Statement	Classification	Notes
<i>collection-statement</i>	Non-contextual	
<i>conditional-statement</i>	Unrestricted contextual	
<i>create-context-statement</i>	Non-contextual	
<i>delete-context-statement</i>	Non-contextual	
<i>rename-context-statement</i>	Non-contextual	
<i>edit-statement</i>	Contextual	Restricted and unrestricted forms
<i>export-statement</i>	Restricted contextual	Some <i>collection</i> restrictions
<i>for-statement</i>	Unrestricted contextual	
<i>output-statement</i>	Unrestricted and unrestricted contextual	Arguments must be consistent with statement usage
<i>set-variables-statement</i>	Unrestricted	
<i>read-statement</i>	Unrestricted	
<i>parameter-statement</i>	Non-contextual	
<i>constrain-statement</i>	Non-contextual	

Common Elements

Most TQL *statements* use one or more instances of common TQL elements. These elements are described in the sections below.

The Variable Element

Variables are named, value-holding *query* elements. Both TQL internal *variables* (TQL Variables and Read Variables) and user-defined *variables* (User Variables) are available. TQL Variables are pre-defined (always present) and their values affect processing of related *statements*. **Table 6** list these *variables*, where they are used, and their effects. Most TQL Variables can be assigned values via the *set-variables-statement*, or through selected command *modifiers*. Read Variables are created as a side-effect of the *read-statement* (see **The Read Statement** section below for further information). User Variables are local to a particular *query* and have names that begin with the percent (“%”) character. User Variables are created and assigned values by the *set-variables-statement*. Once defined, they can subsequently be used in *expressions* (see **The Expression Element** section below).

The Attribute Element

Attributes are elements that refer to objects in the DTS Knowledgebase. *Attributes* include TQL Attributes, described in **Table 2 TQL Attribute Keywords**, DTS element names, direct attributes, and indirect attributes, forms that combine *attributes*.

Direct *attributes* name a DTS Concept or Term Type Attribute, specifically a DTS Synonym Type, Property Type, Role Type or Concept Association Type, or a qualified DTS Property Type or Concept Association Type (using the syntax `PROPORASSN_TYPE.QUALIFIER_TYPE`). Defining Concepts, Defining Roles, Inverse Roles and Inverse Concept Association Types can also be specified. Note that

throughout TQL, caret (“^”) bracketing is required around any explicit reference to the name of an *attribute* that contains any character other than letters, digits and underscores: ^My Property^.

Indirect *attributes* are supported in certain contexts such as *selectors* and *export-statement* arguments. An indirect attribute consists of a Concept-valued source *attribute* (CONCEPT, PARENT, CHILD, ROLE, ASSOCIATION, INV SYNONYM, DEFINING CONCEPT, DTS Role Type, DTS Concept Association Type or inverse DTS Synonym Type), the “referencing” operator “->”, and any string-valued target *attribute* (for example a Concept Property) of the attribute’s source Concept. The resulting combined *attribute* refers to the instance of the target *attribute* associated with the value of the source attribute. For example:

```
PARENT->^Code in Source^
```

refers to the Code in Source Property of the referent Concept’s Parent. Valid target *attributes* for this “referencing” syntax are:

- CONCEPT_NAME or NAME,
- CONCEPT_CODE or CODE,
- CONCEPT_ID or ID,
- CONCEPT_STATUS or STATUS,
- QUALIFIED_NAME,
- PREFERRED_NAME,
- RESOLVED_NAME,
- PRIMITIVE,
- NAMESPACE, and
- Concept and Term Property Type (including anonymous Property Types)

By default, *attribute* references refer to *attributes* in the current *context* namespace. As necessary, an *attribute* can be associated with a specific namespace by placing the *attribute*’s namespace name in brackets after the *attribute* name: ^My Property[MySpace]^ . If the current *context* is a Subset, ConSet, TermSet or All Namespace (ALL WITH ... form) *context*, a non-namespace-qualified *attribute* is called an *anonymous attribute*. The type (actual instance) of an *anonymous attribute* is not resolved until run-time and is determined relative to the namespace of the current concept (or term) in the *collection*. In these *statements*:

```
FROM ALL WITH NAME EQUALS "XY*" EXPORT NAME, ^Code in Source^;  
FROM {MySubset} EXPORT NAME, ^Code in Source^;
```

for each selected concept, the existence of a Code in Source Concept Property Type from the concept’s namespace is determined and if present, it’s value, if any, on the concept is exported.

For an indirect *attribute*, if the source *attribute* references a Concept in a different Namespace, for example, a mapping Concept Association Type, any target Property Type must be qualified with the target Namespace to disambiguate the *attribute*, e.g.:

```
^Map to SNOMED CT^->^Code in Source[SNOMED CT]^
```

Since two DTS Attributes can have the same name, but be of different types, e.g., a Property Type and an Association Type having the same name, TQL provides a syntax to disambiguate these names. The type of an *attribute* can be specified by post-fixing an *attribute* reference with a type code in parenthesis. Thus, `^MyAttribute[Demo]^(CP)` forces `MyAttribute` to be interpreted as a Concept Property Type in the `Demo` Namespace. The following type codes are supported:

P or CP	Concept Property Type
TP	Term Property Type
A or CA	Concept Association
TA	Term Association
SA	Synonym Association
R	Role
PQ or CPQ	Concept Property Qualifier
TPQ	Term Property Qualifier
AQ or CAQ	Concept Association Qualifier
TAQ	Term Association Qualifier

TQL cannot know the type of an *anonymous attribute*, so *anonymous attributes* usually have a type code specified. If no type is specified, Concept Property Type is used as described in the above examples.

The Function Element

Like *variables*, TQL supports both TQL Functions and User Functions. *Functions* take a single argument and return a string value. The *function* argument can be certain TQL Attributes (see **Table 2**) and qualified or unqualified direct *attributes*. Extended *attribute* forms, i.e., the “encoded” and “referencing” forms, are not permitted. *Functions* can be used as *selectors*, *predicates* and in *expressions* when contextual *expression-elements* are permitted. When used as a *selector* or a *predicate*, if a *function* can return multiple values, e.g., `LENGTH(CHILD) EQUALS 3`, an “if any” condition is applied to the *selector* or *predicate* test. **Table 5** describes the available TQL Functions.

A User Function, identified by a *function* whose name begins with a percent sign (“%”), is a user-developed extension to TQL. For example, the `%REVERSE(arg)` function might return the string value of its argument with the characters reversed. A User Function is created by writing a Java class that implements the `TQLFunction` class. This class, packaged in a jar file and placed in the `lib\modules` folder, is recognized by TQL on start-up and can subsequently be used in *expressions* in the same way as a TQL Function. See **Appendix D - Writing a User Function** for further information.

The Expression Element

The *expression* element is used as part of a *selector* in a *collection*, part of a *predicate* in a *conditional-statement*, or in an *edit-statement*, *set-variables-statement*, *export-statement*, *output-statement*, or *constrain-statement*. An *expression* is evaluated at query run-time to produce a string value. Syntactically, an *expression* is a sequence of *expression-elements* separated by *expression-operators*. An *expression-element* can be a numeric literal (12.34), string literal (“confirmed by”), a TQL Variable

(*AXIS*), a Read Variable (*\$1*), a User Variable (*%NAME*), a TQL Attribute (*CONCEPT_NAME*), a DTS Attribute (*^Code in Source^*), a TQL Function (*LENGTH*) or a User Function (*%REVERSE*).

The *expression-operators* and their semantics are:

The plus sign (“+”) Takes the numeric head of each of its arguments and returns the numeric sum (as a string).

The minus sign (“-”) Takes the numeric head of each of its arguments and returns the result of subtracting the second value from the first (as a string).

The ampersand (“&”) Returns the concatenation of the two string values.

The value of an *expression* is the result of applying each *expression-operator* to the run-time values of its right and left *expression-elements*. Evaluation proceeds strictly left to right (grouping via parentheses is not currently supported). Thus:

```
SET %N = 12 + "34AB" & 56;
```

sets the value of User Variable *%N* to the string “ 4656”.

There are two types of *expressions*: contextual and non-contextual. A non-contextual *expression* is one that does not depend on a Concept of Term; it is made up solely of literals and *variables*. Non-contextual *expressions* are typically found in *selectors*. Contextual *expressions* can include an *attribute* or *function*, but there can only be one instance of a contextual element (an *expression-element* based on a Concept or Term), e.g. a TQL Attribute, DTS Attribute or TQL Function, in an *expression*. The following are examples of valid contextual *expressions*:

```
EXPORT "Capital of" & ^State Name^, "is" & ^State Capital^;
```

```
SET ^Code in Source^ = "CIS" & concept_code;
```

Some TQL *statements* have additional restrictions on *expressions*. These are documented in the associated *statement* descriptions.

The Comment Element

TQL permits two types of descriptive *comments* to be interspersed with *statements*. Comments may be inserted at any TQL element position. A line comment begins with two dash characters “--” and continues to the end of the line:

```
FROM myconset WITH CONCEPT_NAME EQUALS "A*" -- this is a line comment
    EXPORT CONCEPT_NAME, CONCEPT_CODE;
```

A block comment is any string between the “/*” and “*/” character pairs:

```
FROM myconset WITH CONCEPT_NAME EQUALS "A*" /* this is a block comment
    that continues to the next line */ EXPORT CONCEPT_NAME, CONCEPT_CODE;
```

The Collection Statement

The *collection-statement* collects a set of Concepts or Terms and, normally, executes a set of *statements* for each Concept (Term) in the resulting *collection*. The *collection-statement* is non-contextual: it cannot be used in the scope of another *collection-statement*.

There is only one form of the *collection-statement*:

FROM [*modifier*] *collection* [*statement-block*] where

$$\textit{statement-block} := \textit{statement} \mid \{ \textit{statement-list} \}$$
$$\textit{statement-list} := \textit{statement} \mid [\textit{statement-list}]$$

<i>modifier</i> :=	“/” CONCEPTS	
	“/” TERMS	
	“/” STATUS “=” <i>status</i>	

```
status := ALL
         ACTIVE
         INACTIVE
         DELETED
```

The optional *collection-statement modifiers* affect the type of *collection* produced by the *statement*. The **CONCEPTS** *modifier* (the default) specifies that the collection should consist of Concepts in the *collection's context*. The **TERMS** *modifier* similarly specifies that the *collection* should be of Terms. At most, one of **CONCEPTS** or **TERMS** may be present. The **STATUS** *modifier* specifies the DTS Status of all the objects (Concepts or Terms) in the *collection*. The default Status is **ACTIVE**. See **The Collection Element** section below for further information.

In most cases, the Concepts or Terms specified in the *collection* are retrieved, then the *statements* in the *collection-statement*'s scope, its *statement-block*, are executed. By the definition, a *statement-block* is either a single *statement*, or a list of statements enclosed in braces. Any *statement* EXCEPT a non-contextual *statement* may be used in the block. See **Table 1** above and the following sections for further information on contextual *statements*. In addition, only one restricted contextual *statement* is permitted in the scope, and, if one is present, there can be no unrestricted contextual *statements*.

The *statement(s)* in the *statement-block* are executed, in the order specified, once for each Concept or Term in the *collection* (for unrestricted contextual *statements*). The *statements* in the *statement-block* are referred to as the subordinate *statements* of the *collection-statement* and their execution state is dependent on that of the enclosing *collection-statement*.

If the optional *statement-block* is not present, the *collection* statement simply enumerates a set of Concepts or Terms. This form is typically used for investigative purposes or for use by external applications using the TQL API. (See ***Using the TQL Class*** section for details on the TQL API.) As an example, the statement below produces no formal output, but the number of Concepts “collected” is

reported in the Console panel of the TQL Editor and is available in the read-only SIZE TQL Variable as described in **The Collection Element** description below.

```
FROM ["SNOMED CT"] WITH ^SNOMED CT to ICD-9-CM map^ EXISTS;
```

The following sections describe additional elements used in the *collection-statement*.

The Collection Element

The *collection* element represents a set of DTS Concepts or Terms upon which TQL actions (*statements*) are performed. The basic *collection* element consists of a *context* (essentially the universe of Concepts or Terms from which *collection* objects are to be drawn) and, optionally, *selectors* that further refine, or filter, the objects to be chosen based on the values of object Attributes. The definition of *collection* is:

```
collection := ALL WITH selectors           |  
              context [ WITH selectors ]      where
```

```
context :=  [" namespace [ ":" version-name ] "]" |  
            [" namespace [ "#" version-date ] "]" |  
            [{" subset [ ":" version-name ] {}" } ] |  
            [{" subset [ "#" version-date ] {}" } ] |  
            conset                               |  
            termset
```

The *selectors* element is defined in the next section.

The *context* of a *collection* can be implicit or explicit. In the first *collection* form above, "ALL" designates that the object context is all Namespaces. In the second form, the context is explicit via the *context* element. In a *context*, *namespace* is the name of an existing DTS Namespace, *subset* is the name of an existing DTS Subset, *authority* is the name of an existing DTS Authority, *conset* is the name of an existing TQL ConSet file, and *termset* is the name of an existing TQL TermSet file, respectively. (A ConSet is a TQL-created file of Concept references and a TermSet is a TQL-created file of Term references. See the description of the *create-context-statement* below for further information on ConSets and TermSets.)

Any *context*, i.e., *namespace*, *subset*, *authority*, *conset* or *termset*, must be entered in the correct case, and, like *attributes*, must be surrounded by caret marks ("^") if the name of the *context* contains any character other than letters, digits and underscores. A Read Variable can be used in the place of a *context* name.

For Namespace and Subset *contexts*, the default Version used is the current (most-recent) Version. As shown above in the definition of *context*, another Version can be specified by providing either the *version-name* (using the colon delimiter) or a snapshot date (using the number-sign delimiter; a number of standard date formats are accepted):

```
[^SNOMED CT^:"2013.07.13AA"]  
[^SNOMED CT^#"12/25/2013"]
```

Only Concepts or Terms from the specified Version will be included in the *collection*.

A TQL *collection* has additional attributes: a *collection-type* (Concepts or Terms), and a *collection-status* (the Status of the Concepts or Terms to be included in the *collection*). These attributes are typically specified as the *modifier* on the FROM command keyword that is placed before the *collection* definition (see above), but can also be supplied explicitly as part of a previous *set-variables-statement*.

The *collection-type* is determined by the presence (or absence) of a CONCEPTS or TERMS *modifier*. The TERMS modifier designates that the objects for the *collection* are DTS Terms. No *modifier* (or the infrequently-used CONCEPTS *modifier*) designates that the *collection* objects are DTS Concepts.

The value of the *collection-status* is set from the STATUS *modifier* or ACTIVE by default. Only concepts/Terms having the *collection-status* are included in the collection.

From these definitions, then, a *collection* can be specified as:

FROM ALL WITH <i>selectors</i>	or
FROM [^SNOMED CT^]	or
FROM/STATUS="INACTIVE" [^SNOMED CT^] <i>selectors</i>	or
FROM/TERMS [^SNOMED CT^] WITH <i>selectors</i>	or
FROM {^CT Procedures^}	or
FROM {^CT Procedures^} WITH <i>selectors</i>	or
FROM myconset	or
FROM myconset WITH <i>selectors</i>	or
FROM/TERMS mytermset WITH <i>selectors</i>	

In the first example, the *collection* is a Concept *collection* whose *context* is all Namespaces and the set of Concepts is completely defined by the *selectors*. (The FROM ALL form places certain restrictions on the selectors. See **The Selector Element** section below for details.) In the second example, the *collection* is all the Concepts in the given Namespace, i.e., all "SNOMED CT" Concepts. In the third example, the *collection* is all the Inactive Concepts from the "SNOMED CT" Namespace (the *context*) that also satisfy the selectors. In the fourth example, the *collection* is a set of Terms from the "SNOMED CT" Namespace which satisfy the *selectors*. The *collection* in the fifth example is all Concepts in the "CT Procedures" Subset (only Concepts can be selected from Subsets). In the sixth example, the "CT Procedures" Subset is the *context* and the set of Concepts is specified by the subsequent *selectors*. The *collection* in the seventh example is simply all the Concepts in the myconset ConSet. (The format for a TermSet is the same.) And in the final two examples, the *context* is the myconset ConSet, or mytermset TermSet, and the *collection* is qualified by the *selectors*.

After a *collection* is evaluated, the read-only TQL Variable SIZE is set to the number of Concepts or Terms in the *collection*. This variable can subsequently be referenced in *output-statements*, etc.

The Selector Element

Selectors further specify the Concepts in the *collection*. *Selectors* consist of one or more *selector* elements combined by the AND, OR and AND_NOT (logical operator) keywords. Parentheses may be

freely used to specify grouping. An example of *selectors* (individual *selector* elements will be described in subsequent sections) would be:

```
^Code in Source^>400 AND  
    (CONCEPT_NAME EQUALS "A*" OR CONCEPT_NAME EQUALS "B*")
```

An individual *selector* is a Boolean expression consisting of a *select-attribute*, a select operator, and (usually) an *expression*. (See the discussion of the specific operators below for details on *expression* requirements.) A *select-attribute* can be a TQL Function or User Function (described in the **Function Element** section above), a TQL Selector Attribute (described in **Table 2**), a direct or indirect *attribute* (described in **The Attribute Element** section above). The Namespace for *selector attributes* defaults to the Namespace specified in the *selector's collection context*. If there is not a Namespace context in the associated *collection*, each *attribute* must be post-fixed with its Namespace: ^Demo Concept [Demo]^ . This form can also be used to override the default Namespace assignment. TQL Variables and *attributes* must also be consistent with the enclosing *collection-type*, e.g., a Term Property Type *selector attribute* can only be used in a Term *collection*. (Entry of *attributes*, including required bracketing, can be simplified through the use of the TQL Attribute Chooser. See the discussion in **The TQL Editor** section below.)

Note that in the ALL WITH ... *collection* form, the *collection* can only consist of *selectors* containing the NAME, CODE, ID or namespace-qualified *attributes*.

The *select-attributes* from each object in the *context* are tested according to the *selectors* and if the result is true, the object is included in the *collection*. Objects testing false are not included in the *collection*. Where multiple occurrences of an *attribute* may be present on an object, for example with Properties, an “if any” condition is implied in evaluating the object for selection.

The Selector Operator Element

Table 3 describes the *selector* operators. As shown in the Table, the various operators may or may not have a right operand, and may interpret the value of the right operand (if required) in different ways.

The two *select-unop* operators, EXISTS and NOT_EXISTS, do not have any right operand. EXISTS selects objects from the *context* on which the associated Attribute exists (has a value). NOT_EXISTS selects objects which do not have the named Attribute. One application of the NOT_EXISTS operator is to find “orphan” objects: Concepts WITH PARENT NOT_EXISTS.

The *select-string-op* operators compare the value of the specified Attribute with the right operand, a non-contextual *expression* (see **The Expression Element** section above for details on *expressions*). If the *select-attribute* is concept-valued (CONCEPT, PARENT, CHILD, ANCESTOR, DEFINING CONCEPT, Role, Concept Association or inverse Synonym), the testing is performed on the Concept Name of the Attribute's target (“to”) Concept. If the *select-attribute* is term-valued (TERM, Term Association, or Synonym), the testing is performed on the Term Name of the Attribute's target (“to”) Term.

For concept and term-valued selectors, the right operand can be a namespace-qualified string:

```
... with ^My Map to SCT^ EQUALS "Event (event) [SNOMED CT]" ...
```

When this form is used, wildcards are not permitted. The namespace reference is removed before testing against the target Concept/Term name, but the resulting target must satisfy the namespace specification. I.e., in the example above, targets whose name is “Event (event)” in namespaces other than SNOMED CT would not satisfy the selector. This form facilitates use of Drag and Drop (from other DTS panels) and TQL *parameters* as the operand.

When a numeric operator (*select-numeric-op*) is used with an Attribute, the *expression* should resolve to a number, and a numeric data type interpretation is made on the leading characters of the specified Attribute value. For example, a selector of `Default_Dose>=4.5` will be satisfied on a `Default_Dose` Property value of “5.2 mg”. Note that the negated value testing operators (`NOT_EQUALS`, `NOT_MATCHES`, and negated numeric operators) only check objects in the *context* which actually have a value for the associated Attribute. To select objects in the *context* that do not have a value for the Attribute, use `NOT_EXISTS`.

The *select-concept-op* operators must be used with concept-valued Attributes (see above) and compare the values of these Attributes with other Concepts as described in the **Table 3**. The *expression* value must resolve to the name of a Concept that represents the “root” of a subtree against whose Concepts the Attribute targets will be tested. Thus:

```
MyMapAssn DESCENDANT_OF "Clinical Finding (finding) [SNOMED CT]"
```

selects all Concepts whose MyMapAssn targets fall under the SNOMED “Clinical Finding (finding)” Concept. It is likely that (as above) the Namespace-qualified Concept name will need to be used in this *selector* since the *context* for the *selector* (i.e., the default Namespace for the Attribute) will usually not be SNOMED CT, but rather another Namespace. Note that all *select-concept-ops* implicitly use the AXIS TQL Variable to determine the hierarchy relationship. See **Table 6** for further information.

The *select-member-op* operator tests whether its left operand (concept-valued) Attribute is a member of the *context* specified by the right operand. This right operand is not an *expression* but a *context* form. Thus:

```
MyMapAssn MEMBER_OF [^SNOMED CT^]
```

selects Concepts whose MyMapAssn targets are in the SNOMED CT Namespace.

There are additional limitations in the combination of the *select-operators* with specific *select-attributes* which are described in detail in **Table 4**. The most important consideration is that some forms currently require searching through all of the objects in the *context*, for example, those with the `NOT_EXISTS` operator or those using an indirect (“referencing”) *attribute* form. This can have severe, if not terminal, effects on performance. As a result, **it is recommended that these forms not be used in large Namespaces such as SNOMED CT**. For further suggestions on the use of *selectors* and *select-operators*, see the **Additional Query Considerations** section below.

Finally, be aware of the interaction between the *collection context* and the *selectors*. The following are examples of *collections* that demonstrate the interaction of these elements:

```
ALL WITH ^Demo Code [Demo]^ EXISTS
```

Collection consists of all Concepts (from any Namespace) that have the Demo Code Property

```
[Demo] WITH ^Demo Code^ EXISTS
```

Collection consists only of Concepts from the Demo Namespace that have the Demo Code Property

```
[^SNOMED CT^] WITH ^Demo Code[Demo]^ EXISTS
```

Collection consists of Concepts from the SNOMED CT Namespace that have the Demo Code Property.

The Conditional Statement

The *conditional-statement* allows execution of subordinate *statement(s)* based on the Boolean value of its associated *predicates* element. The *conditional-statement* is an unrestricted contextual *statement*; it can be used as a subordinate *statement* in a *collection-statement* at any depth.

There are three forms of the *conditional-statement*:

```
IF predicates statement-block  
ELSE statement-block  
ELSEIF predicates statement-block
```

The first (IF) form, takes a single *predicates* argument and has a *statement-block* as a scope (see ***The Collection Statement*** section above for a description of the *statement-block* element). The *predicates* element is similar to the *selectors* element: it has an LHS that is an attribute, a TQL operator, and a RHS Expression. (The *predicates* element is described in detail below.) The *predicates* element is evaluated for each Concept (or Term) in the enclosing *collection*. If the *predicates* evaluates to true, the *statements* in the *statement-block* are executed. If the *predicates* evaluates to false, the *statement-block* is ignored.

```
FROM [Demo]  
  IF NAME EQUALS "T*" PRINT NAME; --print concepts in Demo that begin with 'T'
```

In the above *query*, the equivalent result could be accomplished with a *selector* in the enclosing *collection-statement*. In fact, the use of a *selector* is more efficient than a *predicate* since the *predicate* must be evaluated for every object in the *collection*. Once a *collection* has been created, however, the IF statement can provide new functionality not previously available in TQL.

The second (ELSE) form must immediately follow an IF (or ELSEIF) and inverts the sense of the preceding *predicates*; if the preceding IF (or ELSEIF) *predicates* evaluated to false, the *statements* in the *statement-block* following the ELSE are executed.

The third (ELSEIF) form, which must immediately follow an IF or another ELSEIF, combines the function of an ELSE and IF to more easily permit “chaining” of conditionals. Without this form, nested blocks would be required. Here is an example using all three forms:

```
FROM [^States of the Union^] {
```

```

IF Capital equals "a*" PRINT "The Capital of " & NAME & " starts with A";
ELSEIF Capital equals "b*" PRINT "The Capital of " & NAME & " starts with B";
ELSEIF Capital equals "c*" PRINT "The Capital of " & NAME & " starts with C";
ELSE PRINT "The Capital of " & NAME & " is not an A, B, or C";
}

```

The Predicates Element

The *predicates* element is used in forms of the *conditional-statement* to determine whether or not the *statement's statement-block* is to be executed. The *predicates* element is an extension of the *selectors* element: it consists of one or more individual *predicate* elements combined by the AND, OR and AND_NOT keywords. An individual *predicate* is a Boolean expression consisting of a *predicate-attribute*, a select operator, and (usually) an *expression*. (See **The Selector Element** section above for additional information on these elements.)

A *predicate* differs from a *selector* in three important ways:

1. The *predicate-attribute* permits TQL and User Variables in addition to the DTS Attributes of a *selector*.
2. The *predicate-attribute* can be *anonymous*. See the description of *anonymous-attributes* in the **Attribute Element** section above. In the example below, the predicate accepts concepts whose namespace contains a Code in Source Concept Property Type and whose Code in Source value on the concept (if any) begins with "1":

```
^Code in Source^(CP) EQUALS "1"
```

3. The *expression* of a *predicate* can be contextual – it can contain attributes that are dependent on the Concept or Term of the enclosing *collection*. (See the **Expression Element** section above for details on *expressions*.) Thus the *predicate* below would be permitted in a *conditional-statement*:

```
CODE NOT_EQUALS ^Code in Source^
```

The Create Context Statement

The *create-context-statement* is used to create a named *context*: a DTS Namespace, Authority, Subset, ConSet or TermSet. This *context* can then be used in subsequent TQL *statements*. The *create-context-statement* is a non-contextual *statement*: it cannot be used in the scope of a *collection-statement*.

There are four forms of the *create-context-statement*:

```

CREATE "<" authority ">" ";"
CREATE [" namespace ":" authority [ ":" nstype [ ":" linked_namespace ] ] "]" ";"
CREATE create-context FROM [ /TERMS ] collection ";"
CREATE create-context FROM build-context log-op build-context ";"

```

```
create-context := “{“ subset “:” authority ”}” |
                termset |
                conset
```

```
build-context := “{“ subset ”}” |
                termset |
                conset
```

```
nstype := THESAURUS | ONTYLOG | ONTYLOG_EXTENSION
```

The *namespace*, *subset*, *authority*, *conset* and *termset* elements can be names, string literals, User Variables or Read Variables. The *nstype* and *linked_namespace* elements can be literals, User Variables or Read Variables.

The first form creates a new Authority.

```
CREATE <^My Authority^>;
```

The second form creates a new Namespace. Note that when creating a Namespace, the new Namespace’s Authority must be specified and must be a previously existing Authority. There are four options for this form:

```
CREATE [^My Namespace^:^My Authority^];
CREATE [^My Namespace^:^My Authority^:"THESAURUS"];
CREATE [^My Namespace^:^My Authority^:"ONTYLOG"];
CREATE [^My Namespace^:^My Authority^:"ONTYLOG_EXTENSION":"SNOMED CT"];
```

The first two options are equivalent and create a local Thesaurus Namespace with the specified name and Authority. The third option creates a local Ontylog Namespace, and the last option creates an Ontylog Extension Namespace linked to SNOMED CT as a base Namespace. When creating an Extension Namespace, the linked Namespace name is required and must be enclosed in quotes not carets.

The third form creates a Subset, ConSet or TermSet from the set of Concepts (or Terms) defined by the *collection* element(s). All of the Concepts (or Terms) resulting from the *collection* element are made part of the *create-context*. Note that the *STATUS modifier* is not permitted on the FROM keyword and the TERMS keyword is only permitted when creating a TermSet.

Note: TQL always creates a Subset made up of individual Concepts even if the *collection* is defined using hierarchy attributes such as ANCESTOR or PARENT, i.e., hierarchy-based Subset definition methods are never applied. This may have performance implications for large Subsets. Use the DTS Subset Editor to create true intensional Subset definitions.

An alternate methodology for Subset creation is to set a specific Property on a *collection* using a *set-attributes-statement* (see discussion below) and build the Subset in the DTS Subset Editor with a Subset Expression that uses this Property.

A ConSet (or TermSet) is a named set of Concepts (Terms) that can be used as a *context* in other TQL *statements*. ConSets and TermSets are not a recognized part of the DTS schema and cannot be shared

between different DTS Editor or TQL Editor instances; they are only used for managing local client TQL results. When the features and persistence of DTS Subsets are not required, however, the use of ConSets and TermSets can provide a simpler and more efficient object grouping mechanism. Use of Subsets, ConSets, and TermSets can also sometimes improve the performance of complex queries (see *Additional Query Considerations* below).

Technically, a ConSet or TermSet is implemented as a file in the local file system that contains references to the Concepts/Terms specified by the *context/selector* elements. The name of a ConSet file is *tql_conset.tqc* where *conset* is a TQL *name-word* (a string consisting of letters, numbers and underscores). Similarly, the name of a TermSet file is *tql_termset.tqt*. The file is created in the current working directory.

The fourth form of the *create-statement* creates a Subset, ConSet or TermSet as an algebraic combination of *build-contexts* (Subsets, ConSets or TermSets) The *log-op* connector element is one of the three supported logical operators: OR, AND and AND_NOT. Thus:

```
CREATE consetC FROM consetA OR consetB;
```

places the union (unique sum) of the Concepts in consetA and consetB into consetC;

```
CREATE consetC FROM consetA AND {subsetA};
```

places the intersection (common) Concepts of consetA and subsetA into consetC;

```
CREATE termsetC FROM/TERMS termsetA AND termsetB;
```

places the union of the Terms in termsetA and termsetB into termsetC. When using the terms-type collection (FROM/TERMS) both *build-contexts* must TermSets.

Finally,

```
CREATE {subsetC:myauthority} FROM {subsetA} AND_NOT {subsetB};
```

places those Concepts from subsetA that are not in subsetB into subsetC, i.e., subsetA “minus” subsetB. Note that as of DTS V4, the name of an existing Authority must be added to the *create-context* when creating a subset.

Note: As of TQL 4.0, TQL can create a Subset that consists of Concepts from more than one Namespace. This is only possible when creating the Subset from ConSet(s) that contain Concepts from multiple Namespaces.

The Delete Context Statement

The *delete-context-statement* permits a TQL user to delete DTS and TQL *contexts*. The *delete-context-statement* is a non-contextual statement: it can only be used at the “top-level” of a *query*; it cannot be used as a subordinate *statement*.

The form of the *delete-context-statement* is:

DELETE *context* “;”

The *delete-context-statement* permanently deletes the designated *context*, and its contents, from the DTS Knowledgebase, or file system in the case of a ConSet or TermSet *context*. When executed from the TQL Editor application, a confirmation dialog is first displayed. Successful execution of the *delete-statement* requires that the DTS User has the MANAGE Permission on the affected *context* (for Namespaces, Subsets and Authorities). If the *context* to be deleted is an Ontylog Namespace, no Extension Namespaces can be linked to this namespace.

Note: There is no undo available for the *delete-context-statement*!

The Rename Context Statement

The *rename-context-statement* is used to change the name of a DTS *context*: a Namespace, Subset or Authority. ConSets and TermSets cannot be renamed with this statement. The *rename-context-statement* is a non-contextual statement: it can only be used at the “top-level” of a *query*; it cannot be used as a subordinate *statement*.

There is only one form of the *rename-context-statement*:

RENAME *context* TO *context* “;”

The *context* arguments must be valid Namespace, Subset or Authority context names, and the DTS object associated with the first *context* must exist and be writable. The second context name, must, of course, not already be present.

Examples:

```
RENAME [^My Namespace^] TO [^Your Namespace^];  
RENAME {^My Subset^} TO {^Your Subset^};  
RENAME <^My Authority^> TO <^Your Authority^>;
```

The Edit Statement

An *edit-statement* allows modification of Namespace data in the DTS Knowledgebase. Execution of the *edit-statement* requires that the DTS User has the WRITE Permission on the affected Namespace. The *edit-statement* is a contextual statement: it must be used in the scope of an enclosing *collection-statement*. The *edit-statement* has both restricted and unrestricted forms and some forms have additional restrictions on the enclosing *collection*. Details are provided below.

There are eleven forms of the *edit-statement*:

```
create-concepts-statement:= CREATE_CONCEPTS[/IGNORE_EXISTENCE] create-object-list “;”  
create-terms-statement:=    CREATE_TERMS[/IGNORE_EXISTENCE] create-object-list “;”  
delete-concepts-statement:= DELETE_CONCEPTS  
                             [/IGNORE_EXISTENCE] [/PRUNE_TERMS] [/PERMANENT]  
                             [concept-list] “;”
```

```

delete-trees-statement:=      DELETE _TREES
                               [ /IGNORE_EXISTENCE ] [ /RETAIN_HEAD ] [ /PRUNE_TERMS ]
                               [ /PERMANENT ] “;”
                               DELETE _TREES
                               [ /IGNORE_EXISTENCE ] [ /RETAIN_HEAD ] [ /PRUNE_TERMS ]
                               [ /PERMANENT ] concept-list “;”
delete-terms-statement:=      DELETE _TERMS [ /IGNORE_EXISTENCE ] [ /PERMANENT ] [term-list] “;”

delete-attributes-statement:= DELETE [ /PRUNE_TERMS ] delete-attr-list “;”
set-attributes-statement:=    SET set-attr-list “;”
update-attributes-statement:= UPDATE update-attr-list “;”

delete-attr-list :=           delete-attr-arg [ “,” delete-attr-list ]
set-attr-list :=              ( set-attr-arg | set-var-arg ) [ “,” set-attr-list ]
update-attr-list :=           set-attr-arg [ “,” update-attr-list ]

```

The *create-concepts-statement* and *create-terms-statement* create new instances of Concepts and Terms in the DTS Knowledgebase. These *statements* are restricted contextual *statements*; there can only be one such statement in the scope of the enclosing *collection-statement* and no unrestricted contextual *statements* may be present. In addition, *selectors* are not permitted in the enclosing *collection* of these statements and the *collection-status* must be ALL or ACTIVE. (In the *create-terms-statement*, the use of the *TERMS modifier* is optional.) The *create-object-list* is the list of the names of the objects to be created. String literals, words, attribute forms (^), User Variables, and Read Variables are supported for the names. Optional forms enable specification of the object’s Code and/or Id respectively:

```

CREATE_CONCEPTS ConceptOne, "Concept Two", ^Concept Three^;
CREATE_CONCEPTS Concept1:"CODE1", Concept2:"CODE2";
CREATE_CONCEPTS Concept1::1, Concept2::2;
CREATE_CONCEPTS Concept1:"CODE1":1, Concept2:"CODE2":2;

```

If either Code or Id is not specified, the DTS Server uses its internal Code and Id Generator to provide the required values.

Normally, the *create-concepts-statement* and *create-terms-statement* give an error if any Concept or Term argument already exists in the DTS Knowledgebase. The *IGNORE_EXISTENCE modifier* is available, however, to bypass run-time errors related to the pre-existence of a Concept or Term. This *modifier* has been designed to be used when one of these *statements* is in the scope of a *read-statement*. See **The Read Statement** section below for further information.

The *delete-concepts-statement*, the *delete-trees-statement* and the *delete-terms-statement* delete Concepts and Terms from the DTS Knowledgebase. For the *delete-terms-statement*, if multiple Terms have the same name, all such Terms are deleted. These *statements* are restricted contextual *statements*; there can only be one such statement in the scope of the enclosing *collection-statement* and no unrestricted contextual *statements* may be present.

The default DTS V4 behavior of delete is to set the associated Concept or Term Status to DELETED. If it is desired to permanently delete a Concept or Term, i.e., remove the object from the Knowledge

completely, add the `/PERMANENT` *modifier* to the delete command name. **Once deleted using the PERMANENT modifier, a Concept or Term cannot be recovered.**

These three statements each have two forms: one without arguments and one with arguments. The unargued forms operate on all the Concepts (or Terms) in the enclosing collection. The second, argued forms, operate only on the Concepts or Terms in the argument list. String literals, words, attribute forms (`^^`), User Variables, and Read Variables are supported as arguments.

The *delete-concepts-statement* and *delete-terms-statement* delete the associated (implicit or explicit) Concepts or Terms from the specified Namespace. (The `TERMS` *modifier* is optional in the *delete-terms-statement*.) The *delete-trees-statement* forms are similar to the *delete-concepts-statement* forms except that in addition to deletion of the Concepts specified in the *concept-list*, all of these Concept's descendants are also deleted. (The hierarchy relationship used for descendants is the current value of the `AXIS` TQL Variable. See **Table 6**.) If the `RETAIN_HEAD` TQL Variable (see **Table 6** and the **Set Statement** section below) is true in the context of a `DELETE_TREES` statement, or present as a *statement modifier*, the named (head) Concepts are **not** deleted, only the descendants of the Concepts are removed. When a *delete-concepts-statement*, *delete-terms-statement* or a *delete-trees-statement* is executed from the TQL Editor application, a confirmation dialog is displayed before any deletion occurs.

Normally, the *delete-concepts-statement*, *delete-trees-statement* and *delete-terms-statement* give an error if any Concept or Term argument does not exist in the DTS Knowledgebase. The `IGNORE_EXISTENCE` *modifier* is available, however, to bypass run-time errors related to the non-existence of a Concept or Term. This *modifier* has been designed to be used when one of these *statements* is in the scope of a *read-statement*. See **The Read Statement** section below for further information.

By default, the *delete-concepts-statement* and *delete-trees-statement* only delete Concepts. No Terms associated with any deleted Concepts are removed. If the `PRUNE_TERMS` TQL Variable has been set to `TRUE` (via a previous *set-variables-statement* or command *modifier*), however, Terms that would be made orphans by deletion of the specified Concept will themselves be deleted. (If the `/PERMANENT` *modifier* has been added to the statement, orphan Terms will be permanently deleted as well.) An orphan Term is defined as a Term having no Synonym relationships to any Concepts. See the **Set Statement** section below for further information on the `PRUNE_TERMS` and other TQL Variables.

The *delete-attributes-statement*, *set-attributes-statement* and *update-attributes-statement* operate on the Attributes of the Concepts or Terms specified in the enclosing *collection*. *Selectors* are permitted in the *collection*. These *statements* are unrestricted contextual *statements*; they can be used as subordinate *statements* in a *collection-statement* at any depth.

The *delete-attributes-statement* deletes the specified Attributes on the Concepts or Terms referenced in the *collection* from the DTS Knowledgebase. The *delete-attr-list* is a comma-delimited list of *delete-attr-args*: Synonym Types, Property Types, Defining Concepts, Defining Role Types, and Concept Association Types, or DTS Term Attributes: (inverse) Synonym Types, Term Property Types and Term Association Types. The specified Type can, optionally, be followed by an equals sign ("`=`") and *expression*. When the *expression* is absent, the *delete-attributes-statement* deletes all occurrences of the specified Attributes from the Concepts or Terms referenced in the *collection*. When a value is given, only those Attributes having the value are deleted:

```
DELETE OldProp, RealProp = "false";
```

deletes all occurrences of `OldProp` from all Concepts in the `Demo` Namespace but only those occurrences of `RealProp` that have the value "false".

delete-attr-list elements can also include qualified forms of Property Types and Concept Association Types:

```
DELETE OldProp.Qual;
```

```
DELETE OldProp.Qual = "09/02/2010";
```

```
DELETE OldProp.Qual = "Active". "09/02/2010";
```

The first form deletes all occurrences of the Qualifier `Qual` from all occurrences of the Property Type `OldProp` that exist on any Concept in the `Demo` Namespace. The second form deletes all occurrences of Qualifier `Qual` whose qualifier value is "09/02/2010" from all `OldProp` Property Types. Finally, the third form deletes occurrences of Qualifier `Qual` whose value is "09/02/2010" but only on those `OldProp` Properties that have the property value "Active". When used with Association Types, the first value in the third example refers to the Association's target Concept (or Term).

The `PRUNE_TERMS` TQL Variable can be used on the *delete-attributes-statement* (Concept *collection-types* only) to specify that orphan Terms be removed when Synonyms are deleted. If the `PRUNE_TERMS modifier` is used, the `/PERMANENT modifier` can also be added to the statement to permanently delete the orphan Terms. (See **Table 6** and the *Set Variables Statement* section below for further information.)

The *set-attributes-statement* and *update-attributes-statement* add (or update) Names, Statuses, Synonyms, Properties, Associations, Defining Concepts and Roles on the Concepts or Terms referenced in the *collection*. The *set-attr-list* is a comma-delimited list of elements consisting of `CONCEPT_NAME`, `CONCEPT_STATUS`, `TERM_NAME`, `TERM_STATUS`, `NAME`, `STATUS`, Synonym Types, Concept Property Types, Concept Association Types, Role Types or the keyword `DEFINING_CONCEPT` for Concept forms, or (inverse) Synonym Types, Term Property Types or Term Association Types for Term forms. As with the *delete-attr-list*, an optional the equals sign ("="), and *expression* can be included. Different Types can be mixed in the *set-attr-list*.

Examples:

```
SET MySynonym = "AConcept", MySynonym = "A Concept":P;
```

```
SET batch = 3.0, updated = "set on @D";
```

```
SET INV ^Parent Of^ = "My Root";
```

For Synonym Type and Association Type *set-attr-list* elements, the value must be the name of an existing Term or Concept as appropriate for the context of the statement. To specify that a created Synonym should be the preferred Synonym, append ":P" to the value, as shown in the first example above.

As shown in the second example above, string literals in the value *expression* can contain one or more of the (case-insensitive) TQL special symbols given in **Table 7**. These symbols are replaced by their associated values on *expression* evaluation. In the third example, the `INV` keyword is used to specify that an Inverse Concept Association referencing the “My Root” Concept be set on the selected Concepts.

For Defining Concepts and Role Types, the statement *context* must be an Extension Namespace and the value must be the name of an existing Concept. Note the use of Namespace-qualified arguments to refer to Concepts in the base Ontylog Namespace:

```
SET DEF ^finding site (attribute)[SNOMED CT]^ =  
    "Liver structure (body structure)[SNOMED CT]";  
  
SET DEFINING_CONCEPT =  
    "Liver structure (body structure)[SNOMED CT]";
```

The difference between the *set-attributes-statement* and *update-attributes-statement* is the semantics of how the Synonym, Property or Association Attribute is created. For the *set-attributes-statement*, a new Attribute is always created on each selected object. For the *update-attributes-statement*, a new Attribute is only created if there is not an existing Attribute of the designated Attribute Type on the object. If there is any occurrence of the Attribute Type, all such occurrences will be updated to (replaced by) a single instance of the Attribute Type with the specified value. When an Attribute value is updated, any Qualifiers will be retained. Note that the `CONCEPT_NAME`, `CONCEPT_STATUS`, `TERM_NAME`, and `TERM_STATUS` attributes (or their `NAME` and `STATUS` alias) can only be used with the *update-attributes-statement*.

Similar to the *delete-attributes-statement*, *set-attr-list* elements can include qualified forms:

```
UPDATE batch.date = "@D";  
  
SET ^Parent Of^.Qual = "My Root"."Defined";
```

In the first example, a date Qualifier is updated (or added) with the current date on all existing occurrences of the `batch` Property. If there is no occurrence of a `batch` Property on the Concept, no action is taken. This form (`propType.qualType=expression`) is only available for the `UPDATE` statement. In the second form, applicable to both `SET` and `UPDATE`, a Qualifier `Qual` is updated/set with value “Defined” on all occurrences of the Concept Association Type “Parent Of” whose target is “My Root”. This form will only create a new Concept Association (or Property) if one with the stated value does not already exist.

Setting and updating of Property and Qualifier Attributes respects the presence of any Validator on the respective Property or Qualifier Types. An error is thrown if the specified value does not pass the Validator.

Finally, note that the arguments of the *set-attributes-statement* (the *set-attr-list*) include *set-var-args*, allowing this *statement* to also set TQL *variables*. Both contextual (*attribute*) and non-contextual (*variable*) arguments may be mixed in the *set-attributes-statement*. See the description in the **Set Variables Statement** section below for information on setting the value of *variables*.

The Export Statement

The *export-statement* creates an export file (or list) of DTS Knowledgebase Attributes and their values. Export is performed to the file given in the `EXPORTFILE` TQL variable, or the `Output file` field in the TQL Editor panel if running TQL interactively. The *export-statement* is a restricted contextual *statement*. There can only be one such statement in the scope of the enclosing *collection-statement* and no unrestricted contextual *statements* may be present. Any *collection-type* or *collection-status* can be used, however, some restrictions on the *collection context* may apply.

There are five forms of the *export-statement*:

```
export-concepts-statement:  EXPORT_CONCEPTS [/SUBSET_VIEW] [/TYPEDEFS] “;”  
                             EXPORT_CONCEPTS [/TYPEDEFS] “;”  
export-subset-statement      EXPORT_SUBSET “;”  
export-namespace-statement: EXPORT_NAMESPACE “;”  
export-attributes-statement: EXPORT export-list [ SORTED_BY sort-list ] “;”
```

The first four forms are used to export populated DTS objects from the *collection* of the enclosing *collection-statement* into an XML file. Output can be directed to the console or to files, but only files having an XML extension will be accepted. The paragraphs below describe the general content and structure of the files created from these forms. More specifically, the export files satisfy the `thesaurus-schema-v4`, `extension-schema-v4`, `subset-schema-v4` schemas (.xsd) available in the `bin\tqleditor` directory. For additional details on TQL XML export file processing see the **Exporting to XML** section below.

The first form of the *export-concepts-statement* exports the Terms and Concepts (having the designated Status) of the designated *context* along with all of their Attributes. *Collection selectors* are not permitted with this form. For a Namespace *context*, the export file consists of all Term and Concept elements along with their Attributes. Terms are only included if they are Synonyms of the exported Concepts, and only Attributes owned by the Concept’s Namespace are exported (this handles the case of Subset and ConSet *contexts* for which there is no defined Namespace). Property, Association and Qualifier Type definitions will only be included if the TQL `TYPEDEFS` variable is true or `TYPEDEFS` is included in the command *modifiers*. For all *contexts*, Term and Concept Attribute elements (Synonyms, Properties, Roles, and Concept Associations) are enclosed within their respective Term and Concept elements. If the Namespace is an Ontylog Extension Namespace the Defined-View attributes are exported; non-defining Roles are not included but each Concept’s Primitive Attribute, Defining Concepts and Defining Roles are enclosed.

If the *context* is a Subset and the Subset’s namespaces are Ontylog Namespaces, the `SUBSET_VIEW` *modifier* can be used which exports the “collapsed” Subset Hierarchy Superconcepts/Subconcepts. The Subset Hierarchy reflects the hierarchical relationship of the original Namespace, but “skips” Concepts not in the Subset. The Subset Hierarchy is shown in the DTS Editor Tree panel when a Subset is selected and “Subset Hierarchy” is chosen as the View Axis. One of the objectives of the *export-*

subset-statement with the `SUBSET_VIEW modifier` is to provide a Subset-centric export of Subset Concepts for use by external applications.

The second form of the *export-concepts-statement* exports the Concepts (with their associated Synonym Terms and Attributes) designated by *selectors* present in the *collection*. As with the first form, type definitions are not written unless the TQL `TYPEDEFS` variable is true or `TYPEDEFS` is included in the command *modifiers*.

The *export-concepts-statement* form is a common method for exporting local Concept-based Namespaces. The following Namespace objects are, however, NOT exported by this statement:

- “Mapping” Associations owned by the Namespace but that do not reference any Concept in the Namespace
- Namespace Concept and Term Properties owned by the Namespace but that are associated with Concepts or Terms in other Namespaces
- Namespace Synonyms owned by the Namespace but that are associated with Concepts in Other Namespaces
- Namespace and Namespace Version Properties

See the description of the *export-namespace-statement* below to export these objects.

The *export-subset-statement* exports all the definitional information associated with a Subset. Data on specific members of the Subset is not included. The *context* must be a Subset and *selectors* are not permitted. This form enables exchange of Subset definitions between DTS systems. The *export-subset-statement* provides a complete extract of Subset information including Subset Description, Expression, and TypeDefs and Attributes for Subset and Subset Version Properties. Loading of exported files is supported by Import Wizard V4.1 and later versions.

The *export-namespace-statement* exports all the objects of the designated Namespace. The *context* must be a Namespace and *selectors* are not permitted. Only objects owned by (created in) the Namespace are exported. The export file begins with all Validator, Property, Association and Qualifier Type elements, followed by Concept and Term elements (subject to *STATUS modifier* restrictions) with all of their Attributes, and finally additional elements not associated with Concepts and Terms such as non-local Properties, non-local Associations and Namespace and Namespace Version Properties. If exporting the contents of an Ontylog Namespace, Kind definitions are included. If exporting an Ontylog or Ontylog Extension Namespace, Role Type definitions are included and the Primitive Attribute, Kind Attribute, Defining Concept and Defining Role elements, are enclosed within their associated Concept. Also, Qualifier Attribute elements are enclosed within their associated Property or Association elements. The *export-namespace-statement* produces a complete copy of the Namespace since it includes all objects whether or not they are associated with local Terms or Concepts. Thus this form can be used for any Namespace, including so-called “mapping” Namespaces, which have no Concepts of their own.

Any export file created by either the *export-subset-statement* or *export-namespace-statement* (that includes Type definitions) can subsequently be imported into other DTS Subsets or Namespaces using the Import Wizard plug-in, recreating/reloading the selected contents. It is recommended that such an import be made to a completely empty Namespace or new Subset. Import Wizard Version 4.1 or later MUST BE USED for exports created by TQL Version 4.0. Support for export and import of Ontylog

Namespaces requires TQL Version 4.5 and Import Wizard Version 4.5 respectively. For further information on importing TQL XML files, see the **DTS Import Wizard User Guide**.

The *export-attributes-statement* outputs a set of string values evaluated in the context of each of the Concepts or Terms present in the enclosing *statement's collection*. Optionally, the lines/rows of the export can be sorted by Attributes specified in the *sort-list*. Delimited text (.txt), Excel (.xls and .xlsx) and XML output formats are supported.

Formally, the *export-list* is a comma-delimited list of *expressions*. Arbitrary *expressions* are supported, but most commonly these *expressions* are contextual, containing direct and indirect *attributes*, *functions*, *variables* and the TQL Export Attributes shown in **Table 2**. *Anonymous attributes* are permitted.

If the *attribute* is a Property Type or Concept Association Type, the *attribute's* Qualifiers may also be displayed using the qualifier syntax: `PROPORASSN_TYPE_NAME.QUALIFIER_TYPE_NAME`. If the *attribute* is a Role Type, the form `ROLE_TYPE_NAME.GROUP` returns the Role Group number associated with the Role(s).

For the TQL Export Attributes `SYNONYM`, `PROPERTY`, `ASSOCIATION` and `ROLE`, the default display string is the “value” of the (possibly multiple) DTS Attribute instance(s): Synonym Term Name, Property Value, Association target Concept or Term Name, and Role target Concept Name. To export the Type name of the Attribute, the `TYPE` modifier can be used. The default string is also available using the `VALUE` modifier.

```
EXPORT NAME, PROPERTY.TYPE, PROPERTY.VALUE;
```

An *export-list expression* also supports two extended *attribute* forms: indirect *attributes* and encoded attributes. Often, these forms are used to enable Concept (or Term) Attributes in the *context* Namespace to display Concepts (or Terms) in another Namespace.

Indirect (“referencing”) *attributes* were described in **The Attribute Element** section above. Encoded attributes provide similar indirect functionality, but based on DTSPROPERTY values.

If an export *attribute* is a Property Type, and the value of the Property “encodes” (is the same value as) a Name, Code, Id or unique Property on another Concept, the referenced Concept can be represented using the “encoded” syntax:

```
MAP_PROP|CONCEPT_CODE[^SNOMED CT^]
```

```
MAP_PROP|^Code in Source[SNOMED CT]^
```

In the first example, this creates a concept-valued export element (from the `SNOMED CT` Namespace) using the Concept whose Code is the same value as the base Concept’s `MAP_PROP` Property. (`CONCEPT_NAME`, `CONCEPT_CODE`, and `CONCEPT_ID` can be used in this form.) In the second example, a concept-valued export element is returned using the Concept whose `Code in Source` Property has the same value as the base Concept’s `MAP_PROP` Property. While the explicit Namespace in the second component is not required, as with the referencing syntax it is typically needed since the encoded export element is often in a different Namespace than the statement’s default *context*. By

itself, the exported value of an encoded element will be the referenced Concept's `CONCEPT_NAME`. The encoded form can, however, be combined with the referencing form to access other Attributes as in:

```
MAP_PROP|^Code in Source[SNOMED CT]^->QUALIFIED_NAME
```

If the base Attribute is term-valued (`TERM`, `SYNONYM`, Synonym Type, or Term Association Type) the element displayed in the export record is, by default, the target Term's `TERM_NAME`, but can also be any string-valued Attribute (for example a Term Property) of the base Attribute's target Term. Supported target Attributes are:

- `TERM_NAME` or `NAME` (the default),
- `TERM_CODE` or `CODE`,
- `TERM_ID` or `ID`,
- `TERM_STATUS` or `STATUS`,
- `NAMESPACE`, and
- Term Property Type

Similar considerations as described above to concept-valued forms apply to term-valued forms.

When an *export-attributes-statement* outputs to an XML file, there are constraints on the *export-list*. See the **Exporting to XML** section for further information. The remainder of this section describes considerations of text and Excel export files.

Generally speaking, for text/Excel files the exporter generates one line/row of output for each Concept or Term. In text files, export *expression* values (the results of *export-list* elements) are separated by the value of the TQL field delimiter (the first character of the TQL `DELIMITER` variable); by default the vertical bar or pipe (“|”) character. In Excel, each *expression* value is placed in its own column cell. If on evaluation, an *expression* results in more than one value (e.g., when the associated contextual Attribute has repeated values), however, multiple lines/rows per Concept can be generated. Thus, if Synonym Attribute is specified in an export *expression* and a selected Concept has two synonyms, two lines/rows will be produced, differing only in the value of the `Synonym` field. If more than one field has repeating values, an appropriate number of unique output records will be produced. (The complexity of this multiplicity is the reason that a given *expression* can only have one contextual Attribute.)

TQL differentiates between independent and dependent repeated values. If a specification calls for both: `MyMap->CONCEPT_NAME` and `MyMap.MyMapQualifier` where `MyMap` is a Concept Association Type, and there are two occurrences of the Association (each with one Qualifier) in a Concept, only two rows will be created, corresponding to the two occurrences of the independent Attribute `MyMap`. If one of the `MyMap` Associations had two instances of `MyMapQualifier`, on the other hand, three rows would be produced.

It is sometimes required to be able to place multiple (repeated) Attribute values into a single export field. This “grouping” of values can be accomplished by placing the name of the Attribute to be grouped in parentheses. The Attribute's multiple values are placed in a single export field, with the individual values separated by TQL's (secondary) group delimiter. This delimiter defaults to the colon “:” but can be overridden by setting a second character in the TQL `DELIMITER` variable (see **The Set Statement** section below).

One of the results of grouping is a potential reduction of output rows. Note, however, that some Attribute relationships may be lost, or disrupted, if grouping is specified on multiple related/dependent fields. In the statement below:

```
FROM [Demo] EXPORT CONCEPT_NAME, (MyProp), (MyProp.Qualifier);
```

the second group takes precedence in determining the record count. If a Concept has two `MyProp` Properties, each with two Qualifiers, the output would appear as:

```
... | PropValue1:PropValue2|Qual12:Qual12
... | PropValue1:PropValue2|Qual21:Qual22
```

which may not be what was desired. Without grouping, of course, four rows would be output.

The optional `SORTED_BY` clause specifies alphanumeric sorting of export lines/rows by selected export fields (Attributes). Attributes in the *sort-list* must be (top-level) contextual Attributes from the *expressions* in the *export-list*.

Finally, export operation can be affected by the settings of export-related TQL Variables:

APPEND	If true, export lines are appended to an existing export file. Otherwise, a new file is created.
DATA_FORMAT	Specifies the format (as a Java <code>SimpleDateFormat</code> string) the format of output dates.
DELIMITER	Specifies the primary, and secondary, field delimiters.
EXPORTFILE	Specifies the name of the export file.
HEADER	If true, the default, a field header line is written to the export file.
LIMIT	Limits the number of Concept/Term lines exported. The default value of zero means no limit.
TIME_FORMAT	Specifies the format (as a Java <code>SimpleDateFormat</code> string) the format of output times.
UNIQUE	If true, only unique output lines are exported.

Values for these variables can be set using a *set-variables-statement* or by including the TQL Variable name (and optional value) as a *modifier* on the `EXPORT` command:

```
... EXPORT/HEADER="false"/LIMIT=5/APPEND CONCEPT_NAME, ^Code in Source^;
```

See **Table 6** and the *Set Variables Statement* section below for further information on these effects.

The For Statement

The *for-statement* iterates a Concept or Term object over the instances of an *attribute*. The *for-statement* is an unrestricted contextual *statement*; it can be used as a subordinate *statement* in a *collection-statement* at any depth. *For-statements* may not, however, be nested: a *for-statement* cannot occur within the scope of another *for-statement*.

There is only one form of the *for-statement*:

```
FOR for-attr statement-block
```

where *for-attr* is any TQL or DTS *attribute* that can have multiple values.

TQL *attribute* filtering operations such as *selectors* (WITH ...) and *conditional-statements* (IF ...), have an “if any” interpretation: if any of the *attributes* meet the *predicate* conditions, the Concept or Term is passed to the associated *statement-block* with all of its *attributes*. This makes it impossible to perform operations on specific instances of an *attribute* such as only printing a *Synonym attribute* if its value begins with “A”.

The *for-statement* enables *attribute-specific* operations by passing a Concept or Term object to the associated *statement-block* that only contains one instance of the specified *for-attr*:

```
FROM [^SNOMED CT^] WITH NAME EQUALS "myocardial infarction (disorder)"
    FOR SYNONYM
        IF SYNONYM EQUALS "MI*" PRINT NAME, SYNONYM;
```

The third line of this query is executed once for each Synonym value on the selected Concept. If the Term name of any of these Synonyms begins with “MI”, that Synonym is printed. Note that PRINT, rather than EXPORT, *must* be used in a FOR due to the restricted nature of EXPORT.

As shown above, the *for-statement* is usually used in conjunction with a *conditional-statement* to perform *attribute* filtering.

The *delete-attributes-statement*, *set-attributes-statements* and *update-attributes-statement* can also be used with the *for-statement* to operate on specific instances of attributes:

```
FROM [Demo]
    FOR ^My Prop^
        IF ^My Prop^ EQUALS "old*" DELETE ^My Prop^;
```

Only instances of My Prop whose value begins with “old” will be deleted.

The Output Statement

The *output-statement* enables message strings to be placed in the log and export streams. The *output-statement* is, in general, an unrestricted statement and can appear in any *query* position. Some restrictions do apply to *output-statement* arguments (see below).

The two forms of the *output-statement* are:

```
LOG export-list “;”
PRINT export-list “;”
```

where *export-list* is a comma-delimited list of *expressions* as described above in the **Export Statement** above. TQL *parameters* (see the **Parameterized Queries** section below) can be used in place of the *expressions*. Individual *export-list expressions* must be consistent with the usage of the *output-statement*: if the *statement* is used in a non-contextual position, e.g., as a top-level *statement*, then only non-contextual *expressions* are permitted. When used in a contextual position, any *expression* form is available.

The first form of the *output-statement* writes its arguments (as separate strings) into the log file stream. If the query is being run from the TQL Editor, the strings are also written to the Console panel. Each argument is written on a separate line.

The second form writes its arguments to the TQL export file stream. This form can be used, for example, to place explanatory or separation messages in the export file when multiple *export-statements* are used in a *query*. No header strings are written by PRINT, but a header can be written by a separate PRINT placed before a *collection-statement*. The PRINT form includes delimiters between arguments (the DELIMITER TQL Variable) if the export file is text. If the export file is an Excel file, each argument is written to a different cell.

The Set Variables Statement

The *set-variables-statement* assigns values to *variables*. The *set-variables-statement* is an unrestricted *statement* and can be used anywhere in a *query*. See the **Set Variables** section below for details on *statement* arguments. Note, however, that the contextual *set-attrs-statement* (see the **Edit Statement** section above) accepts both *variable* and *attribute* arguments and as such can perform the same functions as the *set-variables-statement*.

Set Variables

The form of the *set-variables-statement* is:

```
SET set-var-list;
```

where *set-var-list* is a comma-delimited list of *set-var-args*:

```
set-var-arg := variable "=" expression ";;"
```

Both TQL Variables and User Variables may be used in a *set-var-arg*. As described in **The Variable Element** section above, TQL Variables are pre-defined and affect processing of TQL statements, while User Variables have names that begin with the percent ("%") character and are defined by their occurrence in a *set-var-arg*. Once defined, User Variables can be used in any subsequent *expressions*. Both TQL Variable and User Variable names are case-insensitive.

```
SET HEADER = "true", %PROMPT = "Concept name:";
```

Table 6 describes the semantics of each TQL Variable and specifies which *statements* are affected by each *variable* (see the *Applies To* column in **Table 6**). Values set by a *set-var-arg* override internal default values, arguments of the TQL Class `main` method, and values, such as the delimiter, export file

and header, that are specified in the TQL Editor GUI. Values set by the *set-statement* are maintained throughout an instance of the TQL Class and thus may extend across multiple *queries* when the TQL API is used. Since the TQL Editor instantiates a new TQL instance for every *query* execution, however, the TQL Variable values are reset at each run.

A shorthand syntax is also available by which TQL Variables can be set in the context of their associated *statements*. The variables are placed as *modifiers* on the *statement* commands. The *modifier* can just name the variable, in which case the variable's value is set to "true", or a specific value can be given:

```
... EXPORT/HEADER="false"/UNIQUE/EXPORT_FILE="text.txt" ...
```

The availability of a specific *modifier* on a command is *statement*-dependent; not all TQL Variables can be used on all commands. See each *statement* description for details. Note that the scope of these *modifier*-set values is limited to the *statement* itself, i.e., after statement execution, the previous values of all TQL Variables are restored.

The Read Statement

The *read-statement* allows the operations in a *query* to be directed by values in an external file. The *read-statement* is an unrestricted *statement* and can be used anywhere in a *query*. *Read-statements* may not, however, be nested: a *read-statement* cannot occur within the scope of another *read-statement*.

There is only one form of the *read-statement*:

```
READ expression statement-block
```

where *expression* must evaluate at run-time to the name of a .txt, .xls or .xlsx file.

The *read-statement* opens the specified input file, then executes its *statement-block* once for each line, or row, in the file. When each line is retrieved, its fields, or cells if an Excel file row, are assigned to TQL Read Variables: variables whose initial character is '\$'. The value of the first field is given to the variable named \$1, the second to \$2, etc. For text files, fields are determined using the primary delimiter in the TQL DELIMITER variable (see additional details below). Only nine Read Variables are currently supported. Then the *statement-block* is executed.

Read Variables can be used as *context* names, as Concept and Term names, and as part of *expressions*. See the documentation on individual *statements* and elements for further details and limitations.

Here is an example of a query that creates Concepts in the Demo Namespace whose names and codes are taken from fields in an input file:

```
FROM [^Demo^] {  
  READ "load-concepts.xlsx" {  
    LOG "Creating concept "&$1;  
    CREATE_CONCEPTS/IGNORE_EXISTENCE $1:$2;  
  }  
}
```

The first column of the load-concepts.xlsx file contains the desired Concept name, and the second column contains the associated Concept code. Note that a null (empty) value of Concept Code is permitted. Finally, note that the *read-statement* could also have been put outside the *collection-statement*.

As described in **The Edit Statement** section above, the optional IGNORE_EXISTENCE *modifier* tells TQL to ignore any run-time errors related to the pre-existence of a named Concept.

The *read-statement* operation can be affected by the settings of the following TQL Variables:

DELIMITER	Specifies the primary field delimiter for input text files. Ignored for Excel files.
HEADER	If true, the default, the first (header) line of the input file is ignored.
LIMIT	Limits the number of input lines read. This can be of use in testing. The default value of zero means no limit.

Values for these variables can be set using a prior *set-variables-statement* or by including the TQL Variable name (and optional value) as a *modifier* on the READ command:

```
READ/HEADER="false"/LIMIT=5 "my-load-file.xlsx" { ... }
```

Parameterized Queries

TQL *parameters* are special elements in *queries* that can be replaced by user-entered values supplied at run-time. When a *query* containing *parameters* is run in the TQL Editor, a *Parameter Entry Panel* is shown that prompts for the *parameter* values. (See the **TQL Editor** section below for further information on the *Parameter Entry Panel*.) Entered values then replace the *parameter* elements in the *query*. A separate application, the TQL Commander, is also available for running parameterized queries. See the **TQL Commander User Guide** for further information.

Parameters can be used in place of any *expression-element* in a TQL *query*. (The only exception to this rule is that *parameters* cannot replace elements in a *parameter-statement* or *constrain-statement*. See details below.) The form of a *parameter* is the *parameter's* name surrounded by the at-sign (@):

```
FROM ["SNOMED CT"] WITH CONCEPT_NAME EQUALS @Concept Pattern@  
EXPORT CONCEPT_NAME;
```

The *parameter-statement* and *constrain-statement* support the use of *parameters* in *queries*. Both *statements* are non-contextual *statements* and can only be used at the "top-level" of a *query*; they cannot be used as a subordinate *statement*. The *parameter-statement* is required to declare use of a *parameter*. A *parameter-statement* can be placed anywhere in the query, but it must precede any use of the statement's *parameter(s)* in other *statements*. Each argument in the statement declares a parameter and, optionally, its type and help text.

parameter-statement := PARAMETER *parameter-list* ";;"

where *parameter-list* is a comma-delimited list of arguments of the form:

parameter [":" [*parameter-type*] [":" *help-literal*]

The optional *parameter-type* argument provides direction to the *Parameter Entry Panel* for validation of parameter values. The following *parameter-types* are supported:

CONCEPT	The parameter value must be a namespace-qualified name of an existing Concept: "North[States of the Union]"
TERM	The parameter value must be a namespace-qualified name of an existing Term: "Ole Miss[States of the Union]".
INTEGER	Validates the entry value to an integer.
NUMBER	Validates the entry value to a number.
BOOLEAN	Validates the entry value to "true" or "false".
FILE	No validation, but shows a Browse button in the parameter's <i>Parameter Entry Panel</i> row for selecting a file.
STRING	No validation. The default <i>parameter-type</i> .

The optional *help-literal* argument provides a help string to the *Parameter Entry Panel*.

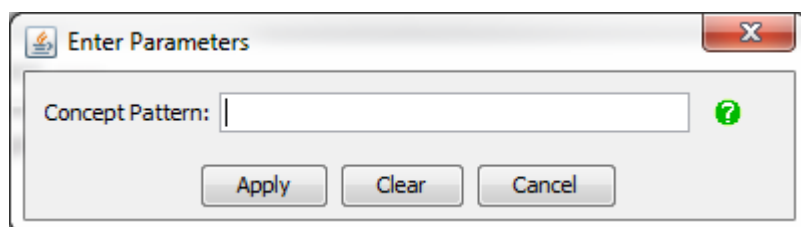
The TQL *Parameter Entry Panel* uses information from the *parameter-statement* and the *constrain-statement* (see below) to construct an entry field and validate the entered value. The prompt label for the field is taken from the name of the *parameter*. The panel constructs a default help string based on the *parameter-type* and any associated constraints, but this default can be overridden by specifying a *help-literal* in the *parameter-statement* argument. The order of appearance of *parameter* fields in the *Parameter Entry Panel* is the order in which the *parameters* are declared in *parameter-statements*.

The following *parameter-statement* would need to precede the example *export-statement* above:

```
PARAMETER @Concept Pattern@::"Enter a concept name search pattern.";
```

Note: *Parameters* cannot be used to replace any *help-literals* in *parameter-statements*.

The screen shot below shows the *Parameter Entry Panel* for this parameter-statement:



The TQL *Parameter Entry Panel* can perform further validations (tests) on input values beyond those associated with the *parameter-type* based on the presence of *constrain-statements* in the *query*:

constrain-statement := CONSTRAIN *parameter* TO *constrain-list* “;”

The *constrain-list* is a comma-delimited list of arguments of the form:

binop operand

where *binop* is any of the binary selector operators shown in **Table 3** and *operand* is the right operand shown in the Table. Each of the arguments in the *constrain-statement* specifies a validation that must be satisfied by any entered value for the designated *parameter*. Value interpretations are performed as required by the specified *binop*, e.g., numeric interpretation for an arithmetic operator, or Concept/Term *context* membership for a MEMBER_OF operator. For the MEMBER_OF operator, the TQL *Parameter Entry Panel* supports implicit “starts with” (wildcard) lookups in the specified *context*.

The *constrain-statement* can appear anywhere in the query after the associated *parameter*’s declaration. All constraints are applied when a value is entered. There is no other dependence on the position of the *constraint-statement*.

The *constrain-statement* below could be used in the example export query:

```
CONSTRAIN @Concept Pattern@ TO NOT_EQUALS "*" ;
```

Note: *Parameters* cannot be used to replace any *operands* in a *constrain-statement*.

Finally, note that the *parameter-statement* and *constrain-statement* are declarative statements used by the TQL Editor (and TQL Commander) subsystems. They are not processed at run-time. In particular, the *constrain-statement* will not validate parameter replacements when queries are run via methods in the TQL Class.

Exporting to XML

A TQL XML export file contains a description of DTS objects associated with a designated Namespace, Subset, or ConSet. If created by an *export-concepts-statement*, *export-subset-statement* or *export-namespace-statement*, all requested objects of the specified *context* are exported formatted according to the associated XML schema definition.

If created by an *export-attributes-statement*, only the selected Concepts and specified *export-attributes* (along with their associated attribute Types) are exported. Because of the highly structured format of an export XML file, restrictions have been placed on the types of *export-attributes* that may be used. In general, the *export-attributes* should be thought of simply as the set of objects to be exported, rather than as describing an output format or order.

Specifically, the following rules must be adhered to in the *export-attributes-statement* when an export is directed to an XML file:

- Export attribute keywords (see **Table 2**) are not permitted in the *export-attributes*. Note, however, that full definitional information for the set of selected Concepts, e.g., Namespace, name, code and id, are always included in the export, and individual Concept Attributes, e.g., Association Types and Property Types, can be specified.
- Qualified Property Types and Association Types are permitted as display attributes. These add the designated Qualifier to the export set.
- Functions are not permitted.
- Any SORTED_BY clause is ignored.

The TQL Editor

TQL queries can be created, maintained, saved, and run from the TQL Editor. The Editor can be started standalone by running `TQLEditor.bat`, or within the DTS Editor by clicking on the TQL icon in the icon bar or selecting `Tools|TQL Editor` from the menu bar. **Figure 1** below shows the layout of the TQL Editor. The *Title Bar* gives information on the status of the current query. The *Menu Bar* gives access to all TQL functions, while the *Icon Bar* provides single click access to frequently-used features. The upper *Editor Area* provides a Notepad-like area for creating and editing TQL queries. See the **Creating and Editing Queries** section below for full details. The middle *Info Area* has two tab panels: the *Console* panel shows status and diagnostic messages, while the *Output* panel is an optional destination for query output. Drag of selected query Concepts from the *Output* panel is supported. See the **Using the Output Panel** section below. Finally, the bottom *Output Specification Area* is used to select the destination and parameters for query output. See the **Specifying Output Location** section below.

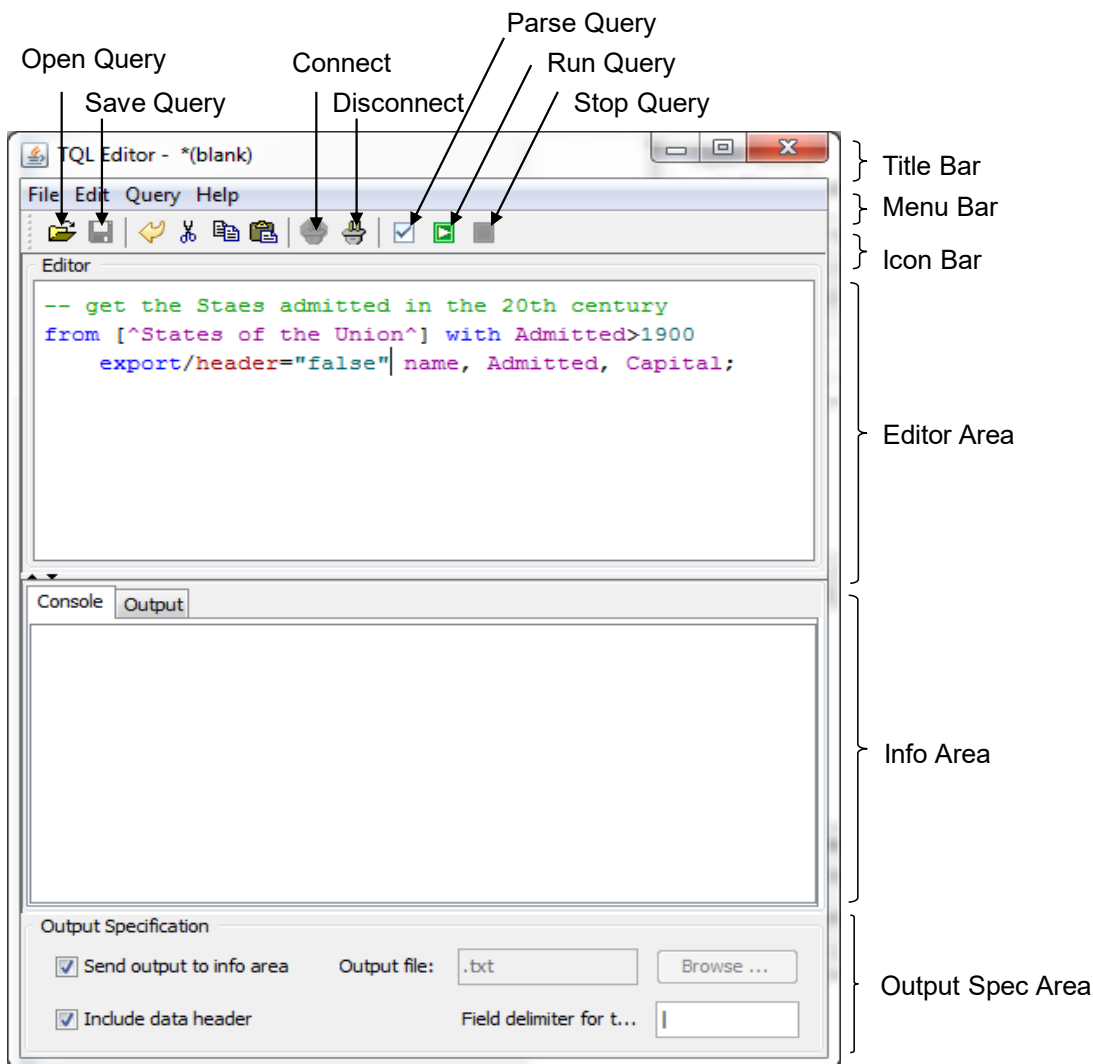


Figure 1 – The TQL Editor

TQL Menus

The File Menu

The File menu contains options for working with TQL files and setting TQL preference values:

New	Clear the <i>Editor Area</i> and current file name in preparation for a new query.
Open Query	Open a file browser dialog to select a saved query.
Save Query	Save the current query in the current file.
Save Query As	Open a file browser dialog to save the query into a new file.
Connect	Connect to a DTS Server (only present when TQL is running in standalone mode).
Disconnect	Disconnect from a DTS Server (only present when TQL is running in standalone mode).
Preferences ...	Open the TQL Preferences dialog. See TQL Preferences below.
Exit	Close the TQL window. If the current query has been modified, and not saved, a warning dialog is shown.

Edit Menu

The Edit menu contains options for editing text in the *Editor Area*:

Undo	Undo the last edit operation.
Cut	Place the selected query text in the Clipboard and erase from the query.
Copy	Place the selected query text in the Clipboard.
Paste	Paste the text contents of the Clipboard into the query at the cursor position.
Clear	Erase all the text in the <i>Editor Area</i> .
Select All	Select all the text in the <i>Editor Area</i> .

Query Menu

The Query menu contains options for processing queries in the *Editor Area*:

Parse Query	Parse the current query.
Run Query	Run (execute) the current query.
Stop Query	Stop the currently executing query.

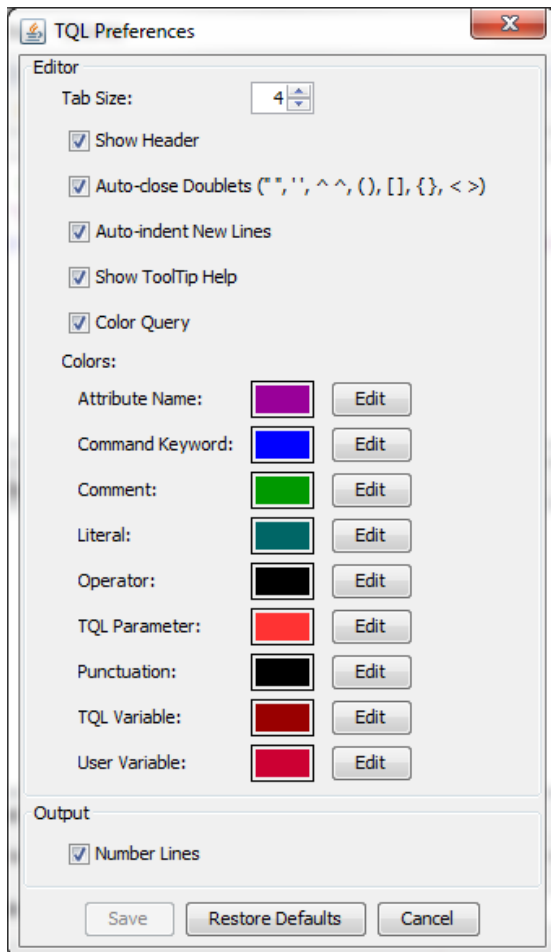
Help Menu

The Help menu contains options for accessing TQL information:

Open TQL User Guide	Opens the TQL User Guide (this document).
Open Attribute Chooser	Opens the TQL Attribute Chooser dialog.
Open RegEx Helper	Opens a TQL RegEx Helper dialog for testing Regular Expressions.
About TQL Editor	Opens the About TQL dialog.

TQL Preferences

The TQL Preferences dialog permits selection of preference items when running TQL. The dialog is shown below:



The upper section of the dialog shows the *Editor Area* preferences while the lower section contains the preference for the *Output* tab of the *Info Area*. Descriptions of the various preference setting are given below:

Editor Preferences

Tab Size	Sets the size (in spaces) associated with a tab character. Tab characters are maintained in the query string. Only the display of the query in the <i>Editor Area</i> is affected by this preference.
Show Header	When selected, enables display of the TQL Header in the <i>Editor Area</i> . See Loading and Saving Queries below for further information on the TQL Header line.
Auto-close Doublets	Certain characters must occur in pairs in TQL queries. This includes single and double quotes, up-arrows, parentheses, brackets, braces, and side-arrow characters. When this preference is enabled, entry of one of

the initial characters of a pair automatically adds the closing character to the text. This enables typing to proceed normally without having to worry about closely the doublet explicitly.

Auto-close Doublets also automatically removes a closing doublet character when the backspace key is pressed in the middle of an empty doublet, e.g., "{}". Empty doublet removal applies to all Auto-close doublets.

Auto-close Doublets is disabled within literals and comments.

Auto-indent New Lines When selected, pressing RETURN in the *Editor Area* will indent the new line (by inserting tab characters) appropriate for the block depth of the query. Additional processing is performed when the RETURN is (1) inside a literal: the literal is closed and restarted on the next line, (2) inside a comment: necessary characters are added to maintain the integrity of the comment, and (3) inside an empty brace doublet: another line is added (with the closing brace) after the new line indented at the original indent level.

Show ToolTip Help When selected, informational ToolTips are shown for the *Command Keyword*, *TQL Variable*, *TQL Modifier* and *Operator* elements. *Command Keyword* ToolTips show all statement forms using the keyword along with any available *Modifiers*. *Variable*, *Modifier* and *Operator* ToolTips give descriptions of the element.

Color Query When selected, the *Editor Area* displays the syntax elements of the query in designated colors. If not selected, the query text color is black. See **Editing Colors** below for information on setting individual element colors.

Output Preferences

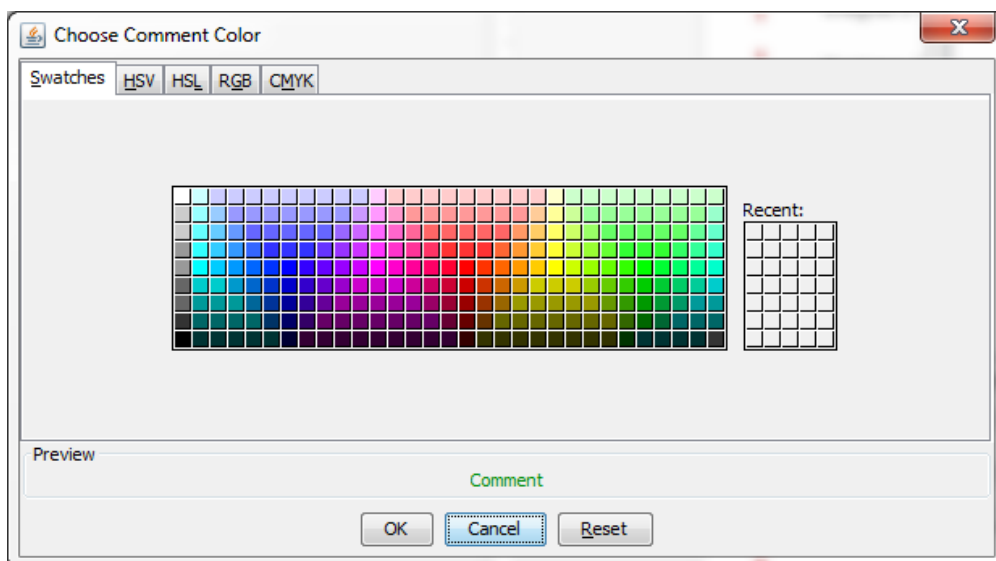
Number Lines When selected, lines in the Output tab of the *Info Area* will be numbered.

After modifying any preference settings, press **Save** to persist the values in the TQL configuration file. Press **Restore Defaults** to set all preference values to their TQL defaults (the new settings will not be automatically saved), or press **Cancel** to exit the TQL Preferences dialog without additional action.

Editing Colors

The **Colors** section of the TQL Preferences dialog enables setting of the text color to be assigned to each syntactic element of a query. Each line in this section (see example above) contains the syntactic element name, e.g. *Comment*, a color box showing the text color to be used and an **Edit** button.

Pressing an **Edit** button opens a **Color Chooser** dialog for the associated element:

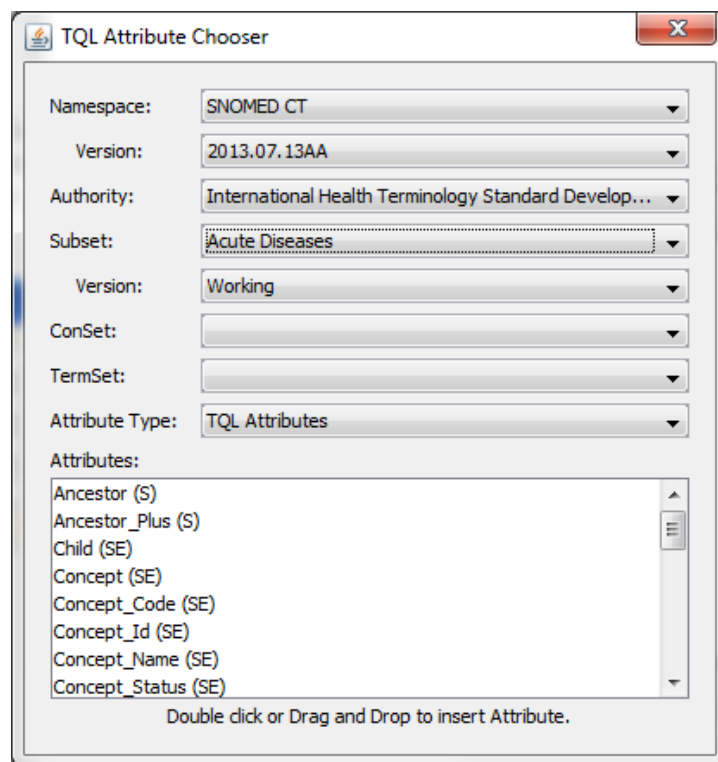


The top of the `Chooser` dialog supports color selection and the preview at the bottom shows the associated element text in the selected color. Multiple tabs are available for color selection: `Swatches` (the simplest), `HSV` (Hue Saturation Value), `HSL` (Hue Saturation Lightness), `RGB`, or `CMYK`. Press the `Reset` button to restore the entry color value or the `OK` button to transfer the color selection back to the `TQL Preferences` dialog.

Creating and Editing Queries

The *Editor Area* is used to create and edit TQL queries. It provides standard Windows editing capabilities (similar to those in Notepad). Editing functions are available from the `Edit` menu (`Undo`, `Cut`, `Copy`, `Paste`, `Clear`, and `Select All`) and the *Icon Bar* (`Undo`, `Cut`, `Copy`, and `Paste`). Standard keyboard and mouse shortcuts are also supported.

To facilitate the entry of DTS Attribute names, the TQL Editor also provides an Attribute Chooser panel (shown at right). To open the panel, right click in the Editor Area. The popup menu has four options: Open Attribute Chooser, Open RegEx Helper, Insert Create Template and Insert Export Template. The first opens the Attribute Chooser Panel, the second opens the RegEx Helper Panel (described in the **Getting Help** section below), and the third and fourth insert the associated “template” query into the *Editor Area*. In the Chooser panel, select the desired Namespace, Subset, ConSet or TermSet and then Attribute Type to show the available Attributes. Attribute Types include TQL Attributes (shown in the example) and TQL Variables, as well as DTS Attributes. In the figure above, TQL Attributes are marked with “E” for Export Attributes and



“S” for Selector Attributes. To insert any Attribute name (or Namespace, Namespace Version, Authority, Subset, Subset Version, ConSet or TermSet name) into the query, drag the name from the Chooser panel to the desired position in the Editor Area (the caret will track the drag position). You can also double click on the element name to replace the current Editor selection (or simply insert at the caret position if there is no selection). The text inserted is the full quoted, namespace qualified, representation of the Attribute.

When the TQL Editor is run as part of the DTS Editor, a Concept (name) can similarly be dragged from other DTS Editor panels into the TQL Editor to populate a value element.

Loading and Saving Queries

Queries can be saved to files (with a “.tql” extension) via the Save Query As option (available in the File menu) and retrieved by the Open Query option (in the File menu and *Icon Bar*). The name of the currently open query file is shown in the TQL Editor *Title Bar*, along with a modified flag (“*”) if appropriate. If the query has not yet been saved, the file name is shown as “(blank)”. A Save option (available in the File menu and *Icon Bar*) rewrites the query into the current file.

Whenever a query is saved, a TQL Header is written at the top of the file. The header is a comment line that gives the TQL version, user and saved date/time:

```
/* TQL V4.0 Query saved by dtsadmin on 2 Sep 2014 21:08:12 */
```

Visibility of the header line in the *Editor Area* can be turned on or off by the Show Header Line item in the File|Preferences menu.

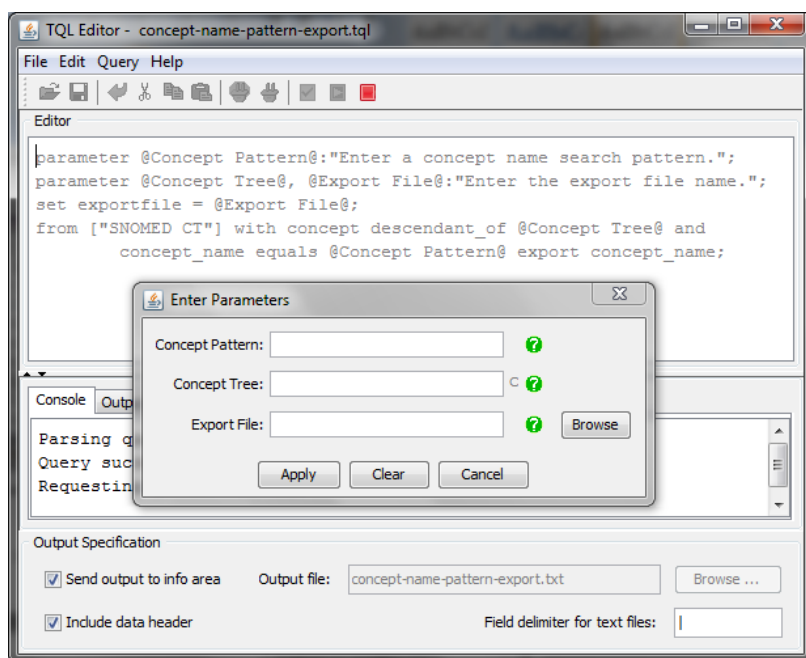
Running Queries

The **Query** menu options (and associated *Icon Bar* items) provide the capability to parse (test/compile) and run queries. Before running (or even parsing) the query, the TQL Editor must, however, be connected to a DTS server. If the TQL Editor is being run as a plug-in to the DTS Editor, this occurs as part of the parent application. If the TQL Editor is being run “stand-alone”, on the other hand, connection must be established explicitly. In this case, **Connect** and **Disconnect** options are available in both the TQL **Query** menu and *Icon Bar*. The **Connect** option opens a dialog box requesting server connect parameters.

The syntactic correctness of a query can be tested by selecting the **Query|Parse Query** menu item (or clicking on the **Parse Query** icon). The query will be compiled and any errors will be reported in the **Console** panel of the *Info Area*. An erroneous token will be highlighted in the *Editor Area*. To parse and run the query, select the **Query|Run Query** menu item or click on the **Run Query** icon. If the query is syntactically correct, it is run and the export output is directed (by default) to the **Output** panel of the *Info Area* (see **Specifying Output Location** below). During the run, status and diagnostic messages are written to the **Console** panel. (Status messages include the elapsed time of the query.) If it is desired to terminate the query run, select the **Query|Stop Query** item or click on the **Stop** icon. This will terminate the query operation, within the procedural granularity provided by the DTS API.

Running Parameterized Queries

When a query is run, if the query has *parameters*, the TQL Editor opens a *Parameter Entry Panel* (see example below) for input of the *parameters*’ values.



The panel displays one line for each *parameter* in the query. The order of the *parameters* in the panel is the order in which the *parameters* are declared in *parameter-statements*. The line's prompt is the name of the *parameter*. The *parameter* value should be entered into the text field. If the *parameter-type* is `CONCEPT`, a small “C” is shown at the right end of the field. This means that the field accepts Drag and Drop of DTS Concepts from other DTS Editor panels. Hover the mouse over the question mark icon to see a help string for the value. A default help string is computed by the panel from the *parameter's* type and context, but a user-defined string can be specified in the *parameter's parameter-statement*. Finally, if the *parameter-type* is `FILE`, a `Browse` button is present. Click this button to open a file browser for selection of an existing file.

Hit `ENTER` to accept the value in a field. The *Parameter Entry Panel* then applies any implicit (*parameter-type*) or explicit (*constrain-statement*) validation rules. After all values have been entered, click on `Apply` to run the query with these values substituted for the *parameter* elements. Alternately, click on `Clear` to clear all existing values, or `Cancel` to terminate query processing.

Specifying Output Location

By default, the query export output is written to the `Output` panel in the `Info Area`. To write the output to a file, uncheck the `Send output to info area` checkbox in the *Output Specification Area* and enter an output file name (or click on the `Browse` button). The output file can be XML (.xml), ASCII text (.txt), or Excel (.xls, .xlsx). See the **Exporting to XML** section above for details and specific constraints on XML output,

For Excel files, a special syntax is available to direct output to a specific sheet of the file. The form:

```
C:\Apelon\MyFolder\concepts.xlsx:First
```

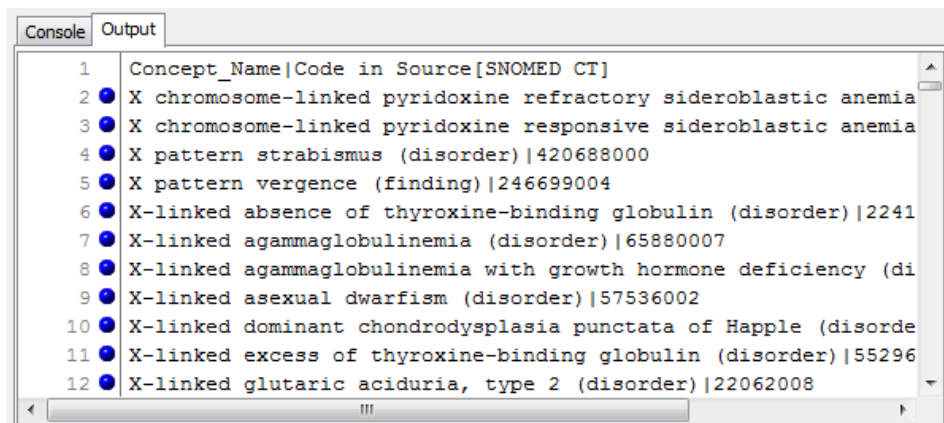
will cause the output to be written to a new sheet in `concepts.xlsx` named `First`. This sheet syntax is only available for .xls and .xlsx files.

For output to a text or Excel file (or the `Output` panel), check the `Include data header` checkbox (see **Figure 1** above) to include an initial header row, and, if a text output, enter the desired field delimiter character(s) in the delimiter text box. The default field delimiter is the vertical bar (“|”) and default group delimiter is the colon (“:”). Alternately, the values of these parameters can be specified using the TQL `EXPORTFILE`, `HEADER`, and `DELIMITER` variables (see **The Set Statement** section above for further details).

Using the Output Panel

In addition to displaying the output from a TQL Query, the `Output` panel can be used as a copy and/or drag source. The example below shows a query result on the right side of the panel with (optional) line numbers and Concept icons on the left. (Numbering of output lines is controlled by a TQL Preference item. See *TQL Preferences* above.) Text can be selected and copied from the right side, while Concept objects can be independently selected and dragged from the left side. A ToolTip on the Concept icon gives the name of the Concept.

Note: When the `UNIQUE` variable is applied to an output query, the Concept associated with an output row is *one of* the Concepts having the unique attribute value. The Concept selected is unspecified.



	Concept_Name Code in Source[SNOMED CT]
1	Concept_Name Code in Source[SNOMED CT]
2	X chromosome-linked pyridoxine refractory sideroblastic anemia
3	X chromosome-linked pyridoxine responsive sideroblastic anemia
4	X pattern strabismus (disorder) 420688000
5	X pattern vergence (finding) 246699004
6	X-linked absence of thyroxine-binding globulin (disorder) 2241
7	X-linked agammaglobulinemia (disorder) 65880007
8	X-linked agammaglobulinemia with growth hormone deficiency (di
9	X-linked asexual dwarfism (disorder) 57536002
10	X-linked dominant chondrodysplasia punctata of Happle (disorde
11	X-linked excess of thyroxine-binding globulin (disorder) 55296
12	X-linked glutaric aciduria, type 2 (disorder) 22062008

Getting Help

The **Help** menu item provides access to this User Guide and an **About** screen for the TQL Editor. The User Guide and **About** are also available in the DTS Editor's **Help** menu.

Also on the TQL **Help** menu is a link to the TQL Regular Expression Helper Panel. This panel provides a testing template and documentation for using Regular Expressions. See Appendix B for further information on Regular Expressions and the Regular Expression Helper Panel.

Additional Query Considerations

TQL includes a rich syntax with which to write queries. Often, this means that a query for a given result, e.g. an export, can be written in multiple ways. This section provides suggestions on reducing errors and improving performance of specific query structures.

A common source of errors is confusion on Attribute value equality in a *selector*. String equality is represented by the `EQUALS` keyword, while the equals character “=” corresponds to numeric equality based on calculation of the numeric head. Use of `EQUALS` (including with wild cards) is typically much more efficient (faster) than numeric equality since the former takes advantage of internal DTS methods while the latter requires an application level search over all Concepts in the current context. As a reminder, string operations have word operators (like `EQUALS` and `MATCHES`) while numeric operations use punctuation operators (“=” and “>”).

Some *select-concept-op* operators duplicate functionality available in other *selector* operations. As an example, the *selectors*:

```
PARENT EQUALS "My Concept"           and
CONCEPT CHILD_OF "My Concept"
```

are equivalent. The two forms are implemented identically. The choice of *selector* form should be made based on *query* clarity.

One way to think of *selector* combinations is that the result of a *selector* is an internal ConSet. Thus the *selectors* expression `selectorA OR selectorB` results in the computation of the union of two ConSets. Similarly `selectorA AND selectorB` results in an intersection computation. In general, the TQL engine executes each *selector* independently. Thus this expression:

```
myprop > 3 AND myprop < 10
```

causes two complete searches of every Concept in the current context.

As mentioned in the discussion of *selectors* in ***The Terminology Query Language*** section, searches over every Concept in a large context, such as SNOMED CT, may result in failed queries due to memory overrun errors on the DTS server or client. For some queries, these errors can be avoided by use of specific query techniques. First, use of an explicit ConSet can remove a potential problem. Thus, if it is desired to see all ICD-9-CM concepts in the `PROCEDURES` tree that have a code less than “80.”, one could use:

```
FROM [^ICD-9-CM^] WITH ANCESTOR EQUALS "PROCEDURES [00-99.99]"
                        AND ^Code in Source^ < 80 EXPORT ... ;
```

The problem is that the second selector must individually search all Concepts in ICD-9-CM, potentially resulting in a fatal error. One solution is to first create a ConSet as:

```
CREATE ICDprocs FROM [^ICD-9-CM^]
                        WITH ANCESTOR EQUALS "PROCEDURES [00-99.99]";
```

This query is safe since the `ANCESTOR EQUALS ... selector` is optimized by the DTS API (see **Table 4**). Now the following export query will only search Concepts in the smaller named ConSet, removing the potential error condition:

```
FROM ICDprocs WITH ^Code in Source[ICD-9-CM]^ < 80 EXPORT ... ;
```

In fact, however, the original query would likely not result in a memory error because the TQL interpreter optimizes `AND` operations in most cases. Specifically, the *selector* on the right of an `AND` operator is applied to the (internal) ConSet resulting from the left *selector* of the `AND` rather than the current context, performing an implicit `AND`. This optimization can be performed unless the *selector* uses the `Ancestor` or `Parent` attribute. Thus the query can be safely executed. Note that reversing the two *selector* elements loses this protection, since the new left *selector* (`^Code in Source^ < 80`) does require a full Concept search. The solution for this structure would be to use an explicit ConSet as described earlier.

Next, consider a query that looks for all “active” ingredient Concepts in NDT-RT:

```
CREATE ActiveIngredients FROM [^NDF-RT^] WITH ^Level^ equals "Ingredient"
AND ANCESTOR NOT_EQUALS "TO BE DELETED INGREDIENT PREPARTATIONS";
```

Since `Ancestor` is used in the right hand *selector*, the `AND` optimization cannot be applied. With the full-concept-search `NOT_EQUALS` operator in the *selector*, a memory error could be generated. The solution is to use the `NOT_AND` connector with the DTS-enabled `EQUALS` *selector* operator:

```
CREATE ActiveIngredients FROM [^NDF-RT^] WITH ^Level^ equals "Ingredient"
AND_NOT ANCESTOR EQUALS "TO BE DELETED INGREDIENT PREPARTATIONS";
```

This query should execute successfully.

Finally, remember that qualifiers are not individually indexed in DTS. Thus any query *selector* which contains a Qualifier Type must retrieve all the instances of the independent Attribute (Property Type or Association Type) and then test any associated qualifiers as specified in the *selector*.

Using the TQL Class

In addition to using the TQL Editor, TQL queries can be executed in two ways: in a standalone, or “batch” mode, using a batch file such as `TQL.bat` (found in the `DTSInstall\bin\tqleditor` folder), or directly from Java applications using the TQL Class. Use of either of these methods avoids GUI overhead which can improve query processing time.

`TQL.bat` provides a simple mechanism to execute previously developed queries as part of routine production processes. `TQL.bat` simply executes the `main()` method of the TQL Class. Use of the `TQL.bat`, or an equivalent batch file, ensures that the DTS context, e.g. `classpath` variable, are set up appropriately.

`TQL.bat` passes required parameters such as connection values and file names to the TQL Class via position-independent key word/value pairs. If no parameters are provided on the `TQL` command, the program will prompt for values from the console. The available parameters are:

<code>-host:hostname</code>	Required. The name of the DTS host (server) system.
<code>-port:portnumber</code>	Required. The port number of the DTS host (server) system.
<code>-instance:instance</code>	Required. The name of the DTS Server instance on the app server.
<code>-user:username</code>	Required. The application server username.
<code>-psw:password</code>	Required. The application server user password.
<code>-query:filename</code>	Required. The name of the TQL query file. Must have an extension of <code>tql</code> .
<code>-output:filename</code>	Optional. The name of the output file. Must have an extension of <code>txt</code> , <code>xml</code> , <code>xls</code> , or <code>xlsx</code> .
<code>-delim:delimiter</code>	Optional. The one or two character delimiter. The first character is the field delimiter and the second character (if present) is the group delimiter. If missing, TQL defaults will apply “ :”.
<code>-header:header</code>	Optional. “Y” to include a header line in export files, or “N” otherwise. If not present, a header is written.
<code>-log:filename</code>	Optional. The name of the log file to be used. If not present, the default TQL log file name, <code>TQL.log</code> , is used.

An example TQL command line is shown below:

```
TQL -user:manager -psw:password -host:localhost -port:4447
-instance:dtjsboss -query:myquery.tql -output:export.txt
-log:"demo log.log"
```

Note the use of quotes around the log file parameter value since the value contains a space. Quote marks can be used around any value but are required if the value contains a space.

For information on using the TQL Class directly by a Java application see the TQL Javadoc file in the `DTSInstall\bin\tqleditor` folder.

Table 2 – TQL Attribute Keywords

<i>Attribute Name</i>	<i>Usage</i>	<i>Description</i>
CONCEPT	EFS	The Concept itself.
CONCEPT_NAME	EFS	The Concept Name.
CONCEPT_CODE	EFS	The Concept Code.
CONCEPT_ID	EFS	The Concept Id.
CONCEPT_STATUS	EFS	The Concept Status.
TERM	EFS	The Term itself.
TERM_NAME	EFS	The Term Name.
TERM_CODE	EFS	The Term Code.
TERM_ID	EFS	The Term Id.
TERM_STATUS	EFS	The Term Status.
NAME	EFS	Alias for CONCEPT_NAME or TERM_NAME.
CODE	EFS	Alias for CONCEPT_CODE or TERM_CODE.
ID	EFS	Alias for CONCEPT_ID or TERM_ID.
STATUS	EFS	Alias for CONCEPT_STATUS or TERM_STATUS.
NAMESPACE	EFS	The name of the Concept's or Term's Namespace.
VERSION_NAME	EFS	The name of the Version for a Concept or Term snapshot.
VERSION_DATE	EFS	The date of the Version for a Concept or Term snapshot.
PREFERRED_NAME	EFS	The Preferred Term (Synonym) of the Concept. Returns the empty string if there is no preferred Synonym.
RESOLVED_NAME	EFS	The Preferred Name if one exists, otherwise the Concept Name
QUALIFIED_NAME	EFS	The ConceptName followed by the Namespace in brackets
PRIMITIVE	EFS	The Primitive Attribute of a Concept in an Ontylog or Ontylog Extension Namespace. The value is "Primitive" or "Defined".
SYNONYM	EF	The Concept's Synonyms.
PROPERTY	EF	The Concept's or Term's Properties.
ASSOCIATION	EF	The Concept's or Term's Associations.
ROLE	EF	The Concept's Roles.
KIND	EFS	The name of the Concept's Kind.
SUBSET	EFS	The name of the Subsets in which the Concept participates.
DEFINING_CONCEPT	EFS	The Defining Concepts associated with a Concept in an Ontylog or Ontylog Extension Namespace.
CHILD	EFS	The Concept's Children (Subconcepts). For Thesaurus Namespaces, the inverse AXIS association is used.
DESCENDANT	EFS	The Concept's Descendants (recursive Subconcepts). For Thesaurus Namespaces, the AXIS association is used.
PARENT	EFS	The Concept's Parents (Superconcepts). For Thesaurus Namespaces, the AXIS association is used.
PARENT_PLUS	S	Same as above but collection includes the Parent Concept.
ANCESTOR	S	The Concept's Ancestors (recursive Parents). For Thesaurus Namespaces, the AXIS association is used.
ANCESTOR_PLUS	S	Same as above but collection includes the Ancestor Concept.

Notes: E = valid as export attribute F = valid as function argument S = valid as selector attribute

Table 3 – TQL Selector Operators

<i>Operator String</i>	<i>Right Operand Type</i>	<i>Decsription</i>
MEMBER_OF	<i>context</i>	Referent Concept is a member of the designated context.
NOT_MEMBER_OF	<i>context</i>	Referent Concept is not a member of the designated context.
CHILD_OF	Concept name literal	Referent Concept is a proper child (Subconcept) of the Concept referenced in the literal. Not permitted in <i>collections</i> whose <i>context</i> is a ConSet.
NOT_CHILD_OF	Concept name literal	Referent Concept is not a proper child (Subconcept) of the Concept referenced in the literal. Not permitted in <i>collections</i> whose <i>context</i> is a ConSet.
CHILD_OF_PLUS	Concept name literal	Same as CHILD_OF but referent Concept can also be equal to the Concept referenced in the literal. Not permitted in <i>collections</i> whose <i>context</i> is a ConSet.
NOT_CHILD_OF_PLUS	Concept name literal	Same as NOT_CHILD_OF but referent Concept can also be equal to the Concept referenced in the literal. Not permitted in <i>collections</i> whose <i>context</i> is a ConSet.
DESCENDANT_OF	Concept name literal	Referent Concept is a descendant (recursive Subconcept/Child) of the Concept referenced in the literal. Not permitted in <i>collections</i> whose <i>context</i> is a ConSet.
NOT_DESCENDANT_OF	Concept name literal	Referent Concept is not a descendant (recursive Subconcept/Child) of the Concept referenced in the literal. Not permitted in <i>collections</i> whose <i>context</i> is a ConSet.
DESCENDANT_OF_PLUS	Concept name literal	Same as DESCENDANT_OF but referent Concept can also be equal to the Concept referenced in the literal. Not permitted in <i>collections</i> whose <i>context</i> is a ConSet.
NOT_DESCENDANT_OF_PLUS	Concept name literal	Same as NOT_DESCENDANT_OF but referent Concept can also be equal to the Concept referenced in the literal. Not permitted in <i>collections</i> whose <i>context</i> is a ConSet.
EQUALS	String literal (wildcards allowed)	String value of Attribute equals the literal.
NOT_EQUALS	String literal (wildcards allowed)	String value of Attribute does not equal the literal.

FOLLOWS	String literal	String value of the Attribute lexically follows the literal.
PRECEDES	String literal	String value of the Attribute lexically precedes the literal.
EXISTS	None	At least on instance of the Attribute exists on the referent Concept. Same as EQUALS *
NOT_EXISTS	None	The Attribute does not exist on the referent Concept. Searches across the full <i>context</i> .
MATCHES	Regular Expression string literal	Attribute value matches the RegEx. See Appendix B for detailed RegEx information.
NOT_MATCHES	Regular Expression string literal	Attribute value does not match the RegEx. See Appendix B for detailed RegEx information.
IN_RANGE	Numeric literal “:” numeric literal	Numeric interpretation of the Attribute value lies between the lower limit and upper limit (inclusive)
NOT_IN_RANGE	Numeric literal “:” numeric literal	Numeric interpretation of the Attribute value does not lie between the lower limit and upper limit (inclusive)
=	Numeric literal	Numeric interpretation of the Attribute value equals the numeric literal.
<>	Numeric literal	Numeric interpretation of the Attribute value does not equal the numeric literal.
<	Numeric literal	Numeric interpretation of the Attribute value is less than the numeric literal.
<=	Numeric literal	Numeric interpretation of the Attribute value is less than or equal to the numeric literal.
>	Numeric literal	Numeric interpretation of the Attribute value is greater than the numeric literal.
>=	Numeric literal	Numeric interpretation of the Attribute value is greater than or equal to the numeric literal.

Table 4 – TQL Selector Attribute/Operator Limitations

<i>Attribute</i>	<i>Operators</i>				
	Concept Op	EQUALS & EXISTS	NOT_EQUALS, PRECEDES, FOLLOWS, MATCHES, & NOT MATCHES	NOT_EXISTS	Numeric
CONCEPT/TERM	None	None	SW, FCS	NA	FCS, NH
CONCEPT_NAME & TERM_NAME	NA	None	SW, FCS	NA	FCS, NH
CONCEPT_CODE, CONCEPT_ID, TERM_CODE & TERM_ID	NA	SW	SW, FCS	NA	FCS, NH
CONCEPT_STATUS & TERM_STATUS	NA	SW, FCS	SW, FCS	NA	FCS, NH
PREFERRED_NAME	NA	SW	SW, FCS	FCS	FCS, NH
PRIMITIVE	NA	FCS	FCS	FCS	FCS, NH
KIND	NA	FCS	FCS	FCS	FCS, NH
VERSION_NAME & VERSION_DATE	NA	FCS	FCS	FCS	FCS, NH
SUBSET	NA	FCS	FCS	FCS	FCS, NH
DEFINING_CONCEPT	None	FCS	FCS	FCS	FCS, NH
PARENT, PARENT_PLUS, CHILD, DESCENDANT, ANCESTOR & ANCESTOR_PLUS	None	None	SW, FCS	FCS	FCS, NH
Synonym Types	NA	None	SW, FCS	FCS	FCS, NH
Property Types	NA	None	SW, FCS	FCS	FCS, NH
Role Types	None	None	SW, FCS	FCS	FCS, NH
Concept Association Types	None	None	SW, FCS	FCS	FCS, NH
Functions	NA	FCS	SW, FCS	NA	FCS, NH

Notes:NA = Not Applicable or Not AllowedSW = Case insensitive, Simplified Wild cards: foo, *foo, *foo*, foo* onlyFCS = Full Concept Search in context; performance implicationsNH = Comparison on Numeric Head**Table 5 – TQL Functions**

<i>Fncion Name</i>	<i>Description</i>
LENGTH	Returns the string length of the Attribute argument.
COUNT	Returns the number of instances of the Attribute argument on the selected Concept.

Table 6 –TQL Variables

<i>Variable</i>	<i>Applies To</i>	<i>Description</i>
APPEND	EXPORT	If “Yes” or “True”, causes export data to be appended to the export file. A new version of the file is not created. The default value is “False”.
AXIS	Hierarchy operations and DELETE_TREES	The name of the Concept Association (or Role in Ontylog Namespaces), used as the parent-to-child “axis” for hierarchy operations. The default is the null string, corresponding to theSuperconcept/Subconcept relationship for Ontylog Namespaces and the “Parent Of” association for Thesaurus Namespaces. The AXIS value can have the “INV” prefix to designate that the actual association points from child to parent.
DATE_FORMAT	<i>expression</i>	The Java SimpleDateFormat format string for outputting dates. The default is “dd- <i>MMM</i> - <i>yyyy</i> ”. See Appendix C for detailed format descriptions.
DELIMITER	EXPORT	The two character delimiter object used in text export operations. The first character is the field delimiter and the second (optional) character is the group delimiter. The TQL default is the vertical bar or pipe “ ” character for the field delimiter and the colon (“:”) for the group delimiter. Overrides the TQL Editor Field delimiter for text textbox and the delimiter argument on the TQL command line.
EXPORTFILE	EXPORT	The name of the desired export file. Overrides the Editor Output file textbox and the output_file_name argument on the TQL command line.
HEADER	EXPORT	“Yes” or “True” to include a header line in the output. The TQL default is “Yes”. Overrides the TQL Editor Include data header checkbox and the header argument on the TQL command line.
LIMIT	EXPORT	If non-zero, limits the number of concept lines exported.TQL default is “0” which means no limit.
PRUNE_TERMS	DELETE_CONCEPTS DELETE_TREES	“Yes” or “True” to delete orphan Terms when Concepts or Synonyms are deleted. The TQL default is “No”.
RETAIN_HEAD	DELETE_TREES	“Yes” or “True” to retain the named, “head”, concepts when trees are deleted. The TQL default is “No”.
SIZE	NA	A read-only variable set to the size of the previous <i>collection</i> .
TIME_FORMAT	<i>expression</i>	The Java SimpleDateFormat format string for outputting times. The default is “h:mm a”. See Appendix C for detailed format descriptions.
TYPEDEFS	EXPORT_CONCEPTS	“Yes” or “True” to include Type Definitions in the export. The TQL default is “False”.

UNIQUE	EXPORT	If “Yes” or “True”, only unique lines are exported. Duplicates are not exported. The TQL default is “False”.
--------	--------	--

Table 7 –Special Symbols for Use in Set/Update Literal Values

<i>Code</i>	<i>Replaced With</i>
@D or @d	The current date. Uses the DATE_FORMAT TQL Variable.
@T or @t	The current time. Uses the TIME_FORMAT TQL Variable

Notes: The values returned from the above symbols are both computed from the current date/time (Java Date()). The separation into a “date” and a “time” is conventional. The actual values returned are completely determined by the format strings.

Revision History

Version 1.0	Initial release.
Version 1.1	Bug fixes. About dialog box. Local Concept Sets (Conssets). <code>CREATE</code> statement. <code>AND</code> optimization. <code>AND_NOT</code> operator. Removed restriction on use of numeric operators. <code>Copy</code> and <code>Select All</code> supported in all text panels. Explicit field delimiter and optional header line to TQL Editor and TQL.
Version 1.2	Support for DTS 3.4. TQL Comments. Subsets as contexts. Synonym Type support added in selector and export expressions. Elapsed time in console status messages. User Guide available in DTS Editor Help Menu and TQL Editor Help Menu. XML output and <i>export-xml-statement</i> . Concept collection process modified to favor collection of large Concept collections over speed (export CASD fetch is delayed). Export Wizard removed. See specific XML export limitations in the Export XML Statement section.
Version 1.3	Support for Ontylog and Ontylog Extension Namespaces (including Defined View variables). Attribute category postfix syntax added.
Version 2.0	<code>Create-statement</code> for Subsets, <code>delete-statement</code> and <code>edit-statement</code> added. Special value codes (@D and @T) added. TQL icon in DTS Editor Icon Bar.
Version 2.1	Minor changes to XML export formats to accommodate addition of XML Schemas.
Version 2.2	New capabilities added for XML exports: fielded exports (via the <i>export-attributes-statement</i>) can be written to XML files, and the <i>export-concepts-statement</i> supports Ontylog Namespaces, Subsets, and Conssets as <i>contexts</i> . The <i>export-subset-statement</i> added for Subset-consistent exports of Subsets. Term Properties and Term Associations are now exported on the <i>export-namespace-statement</i> . DTSPROPERTYTypes are available as a <i>concept-string</i> on exports. An argumented form of the <i>delete-concepts-statement</i> has been added, as has a <i>delete-trees-statement</i> for deletion of concept hierarchies.
Version 3.0	Major update. Multi-statement queries. <code>LOG</code> , <code>PRINT</code> and <code>SET</code> <i>statements</i> as well as new <i>statement</i> forms. TQL <i>variables</i> and <i>modifiers</i> . New <i>selector</i> operators and TQL Attributes. <code>COUNT</code> and <code>LENGTH</code> <i>functions</i> . Encoded expressions. Chooser attributes are now sorted. TQL <code>main()</code> supports socket server connections. See the V3 Release Notes for further details.
Version 3.1	Export to Excel files. <code>CREATE_CONCEPTS</code> and <code>CREATE_TERMS</code> <i>statements</i> . Optional attribute values on <i>delete-attributes-statement</i> . <i>delete-attributes-statement</i> and <i>set-attributes-statement</i> support Synonyms (including <code>PRUNE_TERMS</code> on delete). Grouped export attributes. RegexHelper panel.
Version 3.2	TQL parameters and associated TQLCommander application. <i>Parameter-statement</i> and <i>constrain-statement</i> . Multiple arguments supported in <i>output-statement</i> .

IN_RANGE operator added. Qualifier options on *delete-attributes-statement* and *set-attributes-statement*. Encoded form supports fixed attributes.

- Version 3.3 The *delete-attributes-statement* and *set-attributes-statement* support Defining Roles and Defining Concepts for Extension Namespaces. Bugs fixed in Extension Namespace hierarchy queries. GROUP qualifier for Roles. FOLLOWS and PRECEDES string operators added. The output panel (optionally) displays line numbers and supports drag of Concepts. The File|Preferences menu option selects whether or not output lines are numbered. XML exports of Terms, Term Properties and Term Associations include Term Ids to enable disambiguation of Terms with identical names. The **TQL Reference Guide** pocket trifold added to distribution.
- Version 3.4 Interim update for DTS V4.0. Changes to TQL class constructors and TQLEditor.bat parameters.
- Version 4.0 Major update for DTS V4.0. New capabilities include Term Collections, User Variables, User Functions, and Value Expressions. Version and date qualification of Namespace and Subset Contexts. Status support in Collections. Creation and deletion of Namespaces and Authorities. DATE_FORMAT, TIME_FORMAT and SIZE TQL Variables. New XML export formats support Namespace Properties, Version Properties, and Subset Properties, Subset Version Properties and Subset Expression (compatible with Import Wizard 4.1). Line numbering and Concept/Term drag from Output panel.
- Version 4.1 Query header line, block statements, and conditional (IF, ELSEIF, ELSE) statements added. NAME, CODE, ID, STATUS, PROPERTY, ASSOCIATION, and ROLE TQL Attributes created. SET, PRINT, and LOG arguments extended. **User Guide** restructured and BNF updated.
- Version 4.2 Bug fixes. FOR statement added for iterating over attribute instances. Anonymous attributes introduced for conditional, export and output statements. Anonymous attributes can be used in Subset, ConSet, TermSet and All Namespace contexts.
- Version 4.3 Bug fixes. DTS Layout support updated. DESCENDANT keyword added as a predicate and export attribute.
- Version 4.4 Bug fix for support of Extension Roles in XML exports of Extension Namespaces. Import requires Import Wizard Version 4.4.
- Version 4.5 Update for DTS Version 4.5. Validators supported for Property and Qualifier edits. XML exports include Validators, Kinds, Subset Version Properties, and non-local Synonyms. New export xsds to support Ontylog Namespace exports. Ontylog imports require Import Wizard Version 4.5. Ontylog and Ontylog Extension Namespaces can be created. Kinds are listed in Attribute Chooser. Ability to specify a target “sheet” for Excel output.
- Version 4.5.1 XML Namespace exports support Defined Role Groups.

Version 4.5.2	Performance improvements in XML exports.
Version 4.6	Bug fix for error when exporting non-working versions of subsets.
Version 4.6.1	Performance improvements for Selectors with non-EQUALS operators.
Version 4.7	RENAME statement added. ‘Referencing’ syntax supported in selectors. Hierarchy selectors correctly handle concept name literal that ends in a bracket.
Version 4.7.1	READ statement added. Significant enhancements to the Editor component including syntactic element coloring and “smart” key processing. Bug fixes.
Version 4.7.2	Namespace and Subset exports now include non-local content properties.
Version 4.8	Performance improvements in Namespace and Subset exports. Bug fix in processing of FOR STATEMENT.
Version 4.9	Corrected CREATE and FOR Statement documentation. Update POI jars for Word and Excel processing.

Appendix A - Terminology Query Language Reference

This Appendix gives a formal definition of TQL syntax in a simplified BNF grammar. TQL elements are in *italics*. Keywords are shown in upper case for clarity but are case-insensitive in TQL. Elements are separated by whitespace. The symbol “[]” denotes alternate elements while “[...]” denotes optional elements. Literals may be enclosed in single or double quotes and standard Java character escapes for single quote, double quote, tab and slash are supported. The caret (“^”) can also be escaped so that this character can appear in attribute literals.

query := *statement* [*query*]

statement :=
collection-statement |
conditional-statement |
constrain-statement |
create-context-statement |
delete-context-statement |
rename-context-statement |
edit-statement |
export-statement |
for-statement |
output-statement |
parameter-statement |
read-statement |
set-statement

statement-block := *statement*
“{“ *statement-list* “}”

statement-list := *statement* [*statement-list*]

collection-statement := FROM [*collection-mods*] *collection* [*statement-block*] “;”

collection-mods := *collection-mod* [*collection-mods*]

collection-mod := “/” CONCEPTS |
“/” TERMS |
“/” STATUS “=” *status*

status :=
ALL |
ACTIVE |
INACTIVE |
DELETED

collection := ALL WITH *selectors* |
context [WITH *selectors*]

context := “[“ *namespace* [“:” *version-name*] “]” |
“[“ *namespace* [“#” *version-date*] “]” |

	“{“ subset [“:” version-name] “}” “{“ subset [“#” version-date] “}” “<” authority “>” conset termset	
<i>version-name</i> :=	<i>string-literal</i>	
<i>version-date</i> :=	<i>string-literal</i>	
<i>selectors</i> :=	<i>selector</i> <i>selectors log-op selectors</i> <i>(selectors)</i>	
<i>log-op</i> :=	AND OR AND_NOT	
<i>selector</i> :=	<i>select-attribute unop</i> <i>select-attribute select-string-op expression</i> <i>select-attribute select-member-op context</i> <i>select-attribute select-concept-op expression</i> <i>select-attribute select-numeric-op expression</i> <i>select-attribute select-range-op numeric-literal “:” numeric-literal</i> <i>function select-string-op expression</i> <i>function select-numeric-op expression</i> <i>function select-range-op numeric-literal “:” numeric-literal</i>	
<i>select-attribute</i> :=	CONCEPT CONCEPT_NAME CONCEPT_CODE CONCEPT_ID CONCEPT_STATUS TERM TERM_NAME TERM_CODE TERM_ID TERM_STATUS NAMESPACE NAME CODE ID STATUS PREFERRED_NAME RESOLVED_NAME QUALIFIED_NAME PRIMITIVE KIND SUBSET DEFINING_CONCEPT	

	PARENT	
	PARENT_PLUS	
	ANCESTOR	
	ANCESTOR_PLUS	
	CHILD	
	DESCENDANT	
	<i>DTSSynonymType</i> [“ (SA)”]	
	<i>DTSPROPERTYType</i> [“ (CP)”] [. <i>DTSPROPERTYQualifierType</i> [“ (CPQ)”]]	
	<i>DTSPROPERTYType</i> [“ (TP)”] [. <i>DTSPROPERTYQualifierType</i> [“ (TPQ)”]]	
	[DEF] <i>DTSRoleType</i> [(“R”)]	
	[INV] <i>DTSRoleType</i> [(“R”)]	
	[INV] <i>DTSConceptAssociationType</i> [(“CA”)]	
	[. <i>DTSAssociationQualifierType</i> [(“CAQ”)]]	
	[INV] <i>DSTermAssociationType</i> [(TA)]	
	[. <i>DTSAssociationQualifierType</i> [(“TAQ”)]]	
<i>select-string-op</i> :=	EQUALS	
	NOT_EQUALS	
	MATCHES	
	NOT_MATCHES	
	FOLLOWS	
	PRECEDES	
<i>select-member-op</i> :=	MEMBER_OF	
	NOT_MEMBER_OF	
<i>select-concept-op</i> :=	CHILD_OF	
	NOT_CHILD_OF	
	CHILD_OF_PLUS	
	NOT_CHILD_OF_PLUS	
	DESCENDANT_OF	
	NOT_DESCENDANT_OF	
	DESCENDANT_OF_PLUS	
	NOT_DESCENDANT_OF_PLUS	
<i>select-numeric-op</i> :=	“=”	
	“<”	
	“>”	
	“>=”	
	“<=”	
	“<”	
	“<=”	
<i>select-range-op</i> :=	IN_RANGE	
	NOT_IN_RANGE	
<i>select-unop</i> :=	EXISTS	
	NOT_EXISTS	
<i>conditional-statement</i> :=	IF <i>predicates statement-block</i>	

	ELSE <i>statement-block</i>	
	ELSEIF <i>predicates statement-block</i>	
<i>predicates</i> :=	<i>predicate</i>	
	<i>predicates log-op predicates</i>	
	(<i>predicates</i>)	
<i>predicate</i> :=	<i>selector</i>	
	<i>variable unop</i>	
	<i>variable select-string-op expression</i>	
	<i>variable select-numeric-op expression</i>	
	<i>variable select-range-op numeric-literal</i> “.” <i>numeric-literal</i>	
<i>create-context-statement</i> :=	CREATE “<” <i>authority</i> “>” “;”	
	CREATE “[“ <i>namespace</i> “.” <i>authority</i>	
	[“.” <i>nstype</i> [“.” <i>linked_namespace</i>]] “]” “;”	
	CREATE <i>create-context</i> FROM[/TERMS] <i>collection</i> “;”	
	CREATE <i>create-context</i> FROM <i>build-context log-op build-context</i> “;”	
<i>nstype</i> :=	THESAURUS	
	ONTYLOG	
	ONTYLOG_EXTENSION	
<i>create-context</i> :=	“{ “ <i>subset</i> “.” <i>authority</i> ” }”	
	<i>conset</i>	
	<i>termset</i>	
<i>build-context</i> :=	“{ “ <i>subset</i> ” }”	
	<i>conset</i>	
	<i>termset</i>	
<i>delete-context-statement</i> :=	DELETE <i>context</i> “;”	
<i>rename-context-statement</i> :=	RENAME <i>context</i> TO <i>context</i> “;”	
<i>edit-statement</i> :=	CREATE_CONCEPTS [<i>create-concepts-mod</i>] <i>create-concepts-list</i> “;”	
	CREATE_TERMS [<i>create-terms-mod</i>] <i>create-terms-list</i> “;”	
	DELETE_CONCEPTS [<i>delete-concepts-mods</i>] “;”	
	DELETE_CONCEPTS [<i>delete-concepts-mods</i>] <i>concept-list</i> “;”	
	DELETE_TREES [<i>delete-trees-mods</i>] “;”	
	DELETE_TREES [<i>delete-trees-mods</i>] <i>concept-list</i> “;”	
	DELETE_TERMS [<i>delete-terms-mods</i>] “;”	
	DELETE_TERMS [<i>delete-terms-mods</i>] <i>term-list</i> “;”	
	DELETE [<i>delete-attr-mod</i>] <i>delete-attr-list</i> “;”	
	SET <i>set-attr-list</i> “;”	
	UPDATE <i>update-attr-list</i> “;”	
<i>create-concepts-list</i> :=	<i>concept</i> [“.” <i>string-literal</i> [“.” <i>integer</i>]] [“,” <i>create-concepts-list</i>]	

```

concept [ “:” integer ] [ “,” create-objects-list ]

create-concepts-mod := ”/” IGNORE_EXISTENCE

create-terms-list := term [ “:” string-literal [ “:” integer ] ] [ “,” create-terms-list ] |
term [ “:” integer ] [ “,” create-objects-list ]

create-terms-mod := ”/” IGNORE_EXISTENCE

concept-list := concept [ “,” concept-list ]

term-list := term [ “,” term-list ]

delete-concepts-mods := delete-concepts-mod [ delete-concepts-mods ]

delete-concepts-mod := ”/” IGNORE_EXISTENCE |
“/” PRUNE_TERMS [ “=” boolean-literal ] |
“/” PERMANENT

delete-terms-mods := delete-terms-mod [ delete-term-mods ]

delete-terms-mod := ”/” IGNORE_EXISTENCE |
“/” PERMANENT

delete-tree-mods := delete-tree-mod [ delete-tree-mods ]

delete-tree-mod := “/” AXIS “=” string-literal |
“/” IGNORE_EXISTENCE |
“/” PRUNE_TERMS [ “=” boolean-literal ] |
“/” RETAIN_HEAD [ “=” boolean-literal ] |
“/” PERMANENT

delete-attrs-mod := “/” PRUNE_TERMS [ “=” boolean-literal ]

delete-attr-list := delete-attribute [ “=” expression [ “:” expression ] ] [ “,” delete-attr-list ]

delete-attribute : DTSPROPERTYTYPE [ “(CP)” ]
[ “:” DTSPROPERTYQUALIFIERTYPE [ “(CPQ)” ] ] |
DTSPROPERTYTYPE [ “(TP)” ]
[ “:” DTSPROPERTYQUALIFIERTYPE [ “(TPQ)” ] ] |
DTSSYNONYMTYPE [ “(SA)” ]
DEFINING_CONCEPT
DEF DTSSYNONYMTYPE [ “(R)” ]
[ INV ] DTSCONCEPTASSOCIATIONTYPE [ “(CA)” ]
[ “:” DTSSYNONYMQUALIFIERTYPE [ “(CAQ)” ] ] |
[ INV ] DTSTERMASSOCIATIONTYPE [ “(TA)” ]
[ “:” DTSSYNONYMQUALIFIERTYPE [ “(TAQ)” ] ]

set-attr-list := ( set-attr-arg | set-var-arg ) [ “,” set-attr-list ]

```

<i>set-attr-arg</i> :=	<i>set-attribute</i> "=" <i>expression</i> ["." <i>expression</i>]
<i>set-attribute</i> :=	CONCPT_NAME CONCEPT_STATUS TERM_NAME TERM_STATUS <i>DTSPROPERTYTYPE</i> ["(CP)"] ["." <i>DTSPROPERTYQUALIFIERTYPE</i> ["(CPQ)"]] <i>DTSPROPERTYTYPE</i> ["(TP)"] ["." <i>DTSPROPERTYQUALIFIERTYPE</i> ["(TPQ)"]] <i>DTSSYNONYMTYPE</i> ["(SA)"] DEFINING_CONCEPT DEF <i>DTSRoletype</i> ["(R)"] [INV] <i>DTSCONCEPTASSOCIATIONTYPE</i> ["(CA)"] ["." <i>DTSSOCIATIONQUALIFIERTYPE</i> ["(CAQ)"]] [INV] <i>DTSTERMASSOCIATIONTYPE</i> ["(TA)"] ["." <i>DTSSOCIATIONQUALIFIERTYPE</i> ["(TAQ)"]]
<i>update-attr-list</i> :=	<i>set-attr-arg</i> ["," <i>update-attr-list</i>]
<i>export-statement</i> :=	EXPORT_CONCEPTS [<i>export-concepts-mods</i>] ";" EXPORT_CONCEPTS [<i>export-concepts-mods</i>] ";" EXPORT_SUBSET [<i>export-subset-mod</i>] ";" EXPORT_NAMESPACE [<i>export-namespace-mod</i>] ";" EXPORT [<i>export-mods</i>] <i>export-list</i> [SORTED_BY <i>sort-list</i>] ";"
<i>export-concepts-mods</i> :=	<i>export-concepts-mod</i> [<i>export-concepts-mods</i>]
<i>export-concepts-mod</i> :=	"/" EXPORTFILE "=" <i>string-literal</i> "/" SUBSET_VIEW ["=" <i>boolean-literal</i>] "/" TYPEDEFS ["=" <i>boolean-literal</i>]
<i>export-subsets-mod</i> :=	"/" EXPORTFILE "=" <i>string-literal</i>
<i>export-namespace-mod</i> :=	"/" EXPORTFILE "=" <i>string-literal</i>
<i>export-mods</i> :=	<i>export-mod</i> [<i>export-mods</i>]
<i>export-mod</i> :=	"/" APPEND ["=" <i>boolean-literal</i>] "/" DATE_FORMAT "=" <i>string-literal</i> "/" DELIMITER "=" <i>string-literal</i> "/" EXPORTFILE "=" <i>string-literal</i> "/" HEADER "=" <i>boolean-literal</i> "/" LIMIT "=" <i>integer-literal</i> "/" TIME_FORMAT "=" <i>string-literal</i> "/" UNIQUE ["=" <i>boolean-literal</i>]

for-attr:=

- [INV] SYNONYM
- PROPERTY
- [INV] ASSOCIATION
- ROLE
- PARENT
- CHILD
- DESCENDANT
- DEFINING_CONCEPT
- DTSSynonymType* [“(SA)”]
- DTSPROPERTYType* [“(P)”]
- [DEF] *DTSRoleType* [“(R)”]
- [INV] *DTSRoleType* [“(R)”]
- [INV] *DTSCONCEPTAssociationType* [“(CA)”]
- [INV] *DTSTermAssociationType* [“(TA)”]

$$set-var-list := set-var-arg [\text{,} set-var-list]$$

<i>variable</i> :=	APPEND
	AXIS
	DATE_FORMAT
	TIME_FORMAT
	DELIMITER
	EXPORTFILE
	HEADER
	LIMIT
	PRUNE_TERMS
	RETAIN_HEAD
	SIZE
	TYPEDEFS
	UNIQUE
	"" <i>name-string</i>
	<i>read-variable</i>

$$output\text{-}list := expression \ [\text{“,”} expression \]$$
$$read-mods := read-mod \ [\ read-mods \]$$

<i>read-mod</i> :=	“/” DELIMITER “=” <i>string-literal</i> “/” HEADER “=” <i>boolean-literal</i> “/” LIMIT “=” <i>integer-literal</i>
<i>read-variable</i> :=	”\$” digit-char
<i>parameter-statement</i> :=	PARAMETER <i>parameter-list</i> “;”
<i>parameter-list</i> :=	<i>parameter-arg</i> [“,” <i>parameter-list</i>]
<i>parameter-arg</i> :=	<i>parameter</i> [“:” [<i>parameter-type</i>] [“:” <i>help-string</i>]]
<i>parameter</i> :=	”@” string ”@”
<i>parameter-type</i> :=	CONCEPT TERM INTEGER NUMBER BOOLEAN FILE STRING
<i>help-string</i> :=	<i>string-literal</i>
<i>constrain-statement</i> :=	CONSTRAIN <i>parameter</i> TO <i>constrain-list</i> “;”
<i>constrain-list</i> :=	<i>constrain-arg</i> [“,” <i>constrain-list</i>]
<i>constrain-arg</i> :=	<i>select-string-op</i> <i>expression</i> <i>select-member-op</i> <i>context</i> <i>select-concept-op</i> <i>expression</i> <i>select-numeric-op</i> <i>expression</i> <i>select-range-op</i> <i>numeric-literal</i> “:” <i>numeric-literal</i>
<i>expression</i> :=	<i>expr-element</i> [<i>expr-op</i> <i>expression</i>]
<i>expr-element</i> :=	<i>parameter</i> <i>function</i> <i>select-attribute</i> <i>variable</i> <i>string-literal</i>
<i>expr-op</i> :=	“+” “-” “&”
<i>function</i> :=	COUNT “(” <i>function-arg</i> “)” LENGTH “(” <i>function-arg</i> “)” “%” <i>name-string</i> “(” <i>function-arg</i> “)”

<i>function-arg</i> :=	<i>select-attribute</i> SYNONYM PROPERTY ASSOCIATION ROLE
<i>export-list</i> :=	<i>export-arg</i> [, <i>export-list</i>]
<i>export-arg</i> :=	“(” <i>export-expr</i> ”)” <i>export-expr</i>
<i>export-expr</i> :=	<i>export-expr-element</i> [<i>expr-op</i> <i>export-expr</i>]
<i>export-expr-element</i> :=	<i>expr-element</i> CONCEPT “->” <i>concept-disp-attr</i> TERM “->” <i>term-disp-attr</i> SYNONYM [“.” TYPE] SYNONYM [“.” VALUE SYNONYM “->” <i>term-disp-attr</i> INV SYNONYM “->” <i>concept-disp-attr</i> PROPERTY [“.” TYPE] PROPERTY “.” VALUE ASSOCIATION [“.” TYPE] ASSOCIATION “.” VALUE ROLE [“.” TYPE] ROLE “.” VALUE PARENT “->” <i>concept-disp-attr</i>] CHILD “->” <i>concept-disp-attr</i> DESCENDANT “->” <i>concept-disp-attr</i> DEFINING_CONCEPT “->” <i>concept-disp-attr</i> DTSSynonymType [“(SA)”] “->” <i>term-disp-attr</i> DTSPROPERTYType [“(P)”] [“” <i>encode-attr</i> [->” <i>concept-disp-attr</i>] DTSPROPERTYType [“(P)”] [“”DTSPROPERTYType [“(P)”] [->” <i>concept-disp-attr</i>]] [DEF] DTSPROPERTYType [“(R)”] [.GROUP] [INV] DTSPROPERTYType [“(R)”] [.GROUP] [DEF] DTSPROPERTYType [“(R)”] “->” <i>concept-disp-attr</i> [INV] DTSPROPERTYType [“(R)”] “->” <i>concept-disp-attr</i> [INV] DTSPROPERTYType [“(CA)”] “->” <i>concept-disp-attr</i> [INV] DTSPROPERTYType [“(TA)”] “->” <i>term-disp-attr</i>
<i>encode-attr</i> :=	CONCEPT_NAME [[“ namespace “]] CONCEPT_CODE [[“ namespace “]] CONCEPT_ID [[“ namespace “]]
<i>concept-disp-attr</i> :=	CONCEPT_NAME CONCEPT_CODE CONCEPT_ID CONCEPT_STATUS NAMESPACE

	NAME		
	CODE		
	ID		
	STATUS		
	PRIMITIVE		
	KIND		
	PREFERRED_NAME		
	QUALIFIED_NAME		
	RESOLVED_NAME		
	DTSPROPERTYTYPE [“(CP)”]		
<i>term-disp-attr</i> :=	TERM_NAME		
	TERM_CODE		
	TERM_ID		
	TERM_STATUS		
	NAME		
	CODE		
	ID		
	STATUS		
	DTSPROPERTYTYPE [“(TP)”]		
<i>sort-list</i> :=	<i>sort-attribute</i> [, <i>sort-list</i>]		
<i>sort-attribute</i> :=	<i>concept-string</i>		<i>sort-attribute</i> must exist in <i>export-list</i>
	SYNONYM		
	PROPERTY		
	ASSOCIATION		
	ROLE		
	PARENT		
	CHILD		
	DESCENDANT		
	DTSSYNONYMTYPE [“(SA)”]		
	DTSPROPERTYTYPE [“(CP)”]		
	DTSPROPERTYTYPE [“(TP)”]		
	DTSROLETYPE [“(R)”]		
	DTSCONCEPTASSOCIATIONTYPE [“(CA ”]		
	DTSCONCEPTASSOCIATIONTYPE [“(TA)”]		
<i>string-literal</i> :=	“ string “		
	‘ string ‘		
<i>numeric-literal</i> :=	<i>parameter</i>		
	integer [”.” [integer]]		
	” integer [”.” [integer]] ”		
	’ integer [”.” [integer]] ’		
<i>integer-literal</i> :=	<i>parameter</i>		

	integer			
	" integer "			
	' integer '			
<i>boolean-literal</i> :=	<i>parameter</i>			
	"TRUE"			
	'TRUE'			
	"FALSE "			
	'FALSE'			
<i>namespace</i> :=	<i>name-string</i>		"^" string "^"	<i>variable</i>
<i>linked_namespace</i> :=	<i>name-string</i>		"^" string "^"	<i>variable</i>
<i>subset</i> :=	<i>name-string</i>		"^" string "^"	<i>variable</i>
<i>authority</i> :=	<i>name-string</i>		"^" string "^"	<i>variable</i>
<i>conset</i> :=	<i>name-string</i>		"^" string "^"	<i>variable</i>
<i>termset</i> :=	<i>name-string</i>		"^" string "^"	<i>variable</i>
<i>concept</i> :=	<i>name-string</i>		"^" string "^"	<i>string-literal</i> <i>variable</i>
<i>term</i> :=	<i>name-string</i>		"^" string "^"	<i>string-literal</i> <i>variable</i>
<i>name-string</i> :=	<i>name-char</i> [<i>name-string</i>]			
<i>name-char</i> :=	alpha-char digit-char "_"			
<i>integer</i> :=	digit-char [<i>integer</i>]			
<i>comment</i> :=	-- comment text to end of line /* comment text */			

Appendix B - Regular Expression Primer

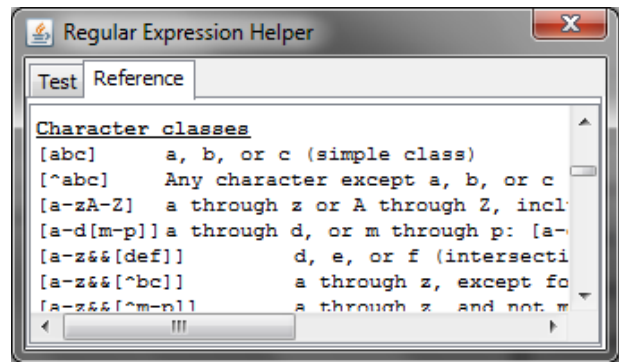
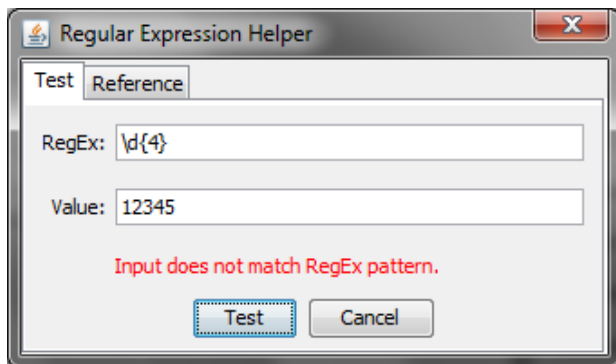
Regular Expressions are a technique to match and/or manipulate string values. Using a Regular Expression (or 'RegEx'), it is possible to search for attributes whose value matches a pattern, validate that an input value matches a given pattern, or even edit a value based on a pattern. TQL supports the first use-case through its MATCHES operator. The Import Wizard Module supports the second and third use-cases via its Parameter Filter functions.

RegExes are very powerful and potentially confusing. At one level, a RegEx pattern is simply a text string, but this string can incorporate a number of special formats that enable quite sophisticated matching. To test for a valid email address, for example, you could use the RegEx pattern:

```
[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}
```

The objective of this Primer is to describe the basic elements of RegExes and show how they can be used for simple pattern matching in TQL. By the end of the Primer, you should be able to explain the RegEx above! For a comprehensive discussion of regular expressions, see the regular expression tutorial at <http://www.regular-expressions.info/>.

Follow along with the examples below by opening the TQL **RegEx Helper** dialog (shown below). This dialog lets you test RegEx expressions and review the RegEx Reference document.



Matching Literals

In the simplest RegEx patterns, the characters stand for themselves. So to see if a value equals "abc" in TQL, use the following (since the RegEx is itself a string, it must be surrounded in quotes):

```
value MATCHES "abc"
```

As will be discussed below, certain characters have special meanings and cannot be used literally in a RegEx. The special characters are the backslash '\', the caret '^', the dollar sign '\$', the period or dot '.', the vertical bar or pipe symbol '|', the question mark '?', the asterisk or star '*', the plus sign '+', the opening parenthesis '(', the closing parenthesis ')', the opening square bracket '[', and the opening curly brace '{'. To match these characters literally in a RegEx, they must be preceded ("escaped") with a backslash. To match the string "\$15.50", for example, use:

```
value MATCHES "\\$15\\.50"
```

Character Classes

Character classes are a way to denote that one of a number of characters is permissible for a match. In a RegEx, you describe a character class by surrounding the class “definition” with square brackets. To match an ‘a’ or an ‘e’, use ‘[ae]’. The RegEx ‘gr[ae]y’ would match either ‘gray’ or ‘grey’. Note that a character class matches only a *single* character; ‘gr[ae]y’ does not match ‘graay’ or ‘greay’ or any other string. The order of the characters inside a character class does not matter.

You can use a hyphen inside a character class to specify a range of characters: ‘[0-9]’ matches a *single* digit between 0 and 9. You can include more than one range in a class: ‘[0-9a-fA-F]’ matches a single hexadecimal digit, case insensitively. And you can combine ranges and single characters: ‘[0-9a-fXA-F]’ matches a hexadecimal digit or the letter X.

A caret after the opening square bracket negates the character class. The result is that the character class matches any character that is *not* in the character class. So the RegEx ‘q[[^]x]’ matches the ‘qu’ in ‘question’, but does *not* match the ‘q’ in ‘qxyz’ or ‘Iraq’ (since there is no character after the q for the negated character class to match).

Predefined Character Classes

RegExes support a number of predefined character classes to simplify writing of patterns. The most common are:

.	matches any character
\d	matches a digit: [0-9]
\D	matches a non-digit: [^0-9]
\s	matches a whitespace character: [\t\n\x0B\f\r]
\S	matches a non-whitespace character: [^\s]
\w	matches a word-character: [a-zA-Z_0-9]
\W	matches a non-word character: [^\w]

Use ‘abc\d’ to match ‘abc1’, ‘abc2’, etc., and ‘\w=3’ to match ‘a=3’, ‘B=3’, and ‘6=3’.

Character Repetition

RegExes contain elements called “quantifiers” to signify that that a character (or character class) can occur multiple times. If ‘X’ is a character or character class, the quantifiers are:

X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
X{n}	X, exactly n times
X{n,}	X, at least n times
X{n,m}	X at least n but not more than m times

Note the difference between square and curly brackets. Square brackets denote a character class, while curly brackets (braces) are for repetition.

Some examples:

‘\w+=\d+’ matches ‘var=34’ or any similar string.

‘abc.*’ is the same as *starts with ‘abc’*, while:

`'.*abc'` is *ends with* `'abc'`, and

`'.*abc.*'` is *contains* `'abc'`

`'[1-9][0-9]{3}'` matches a number between 1000 and 9999 and

`'[1-9][0-9]{2,4}'` matches a number between 100 and 99999.

Alternation and Grouping

Whole RegEx patterns can be alternated and grouped. To specify alternate RexEx patterns, use the pipe character. The RegEx `'cat|dog|fish'` matches “cat” or “dog” or “fish”. Alternation has the lowest precedence of all RegEx operators so use parenthesis for grouping:

`'cat|dog food'` matches “cat” or “dog food”, but

`'(cat|dog) food'` matches “cat food” and “dog food”

Warning

Finally, be careful about the use of the backslash. Remember that backslashes in literals need to be escaped. The *RegEx Helper* takes care of this automatically, so the examples in this Primer can be entered as shown, but when writing TQL queries, always double backslashes. To use the last example in the **Matching Literals** section, use:

```
value MATCHES "\\$15\\.50"
```

and to test that a value that contains a backslash, the following is required:

```
value MATCHES "\.*\\\\\.*"
```

Appendix C - Date Format String

Date and time formats are specified by *date and time pattern* strings from the Java `SimpleDateFormat` class. The discussion below provides a simplified description of the format string. Within date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using single quotes (') to avoid interpretation. "'" represents a single quote. All other characters are not interpreted; they're simply copied into the output string during formatting or matched against the input string during parsing.

The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved):

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
Y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800

Pattern letters are usually repeated, as their number determines the exact presentation:

- **Text:** For formatting, if the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used if available. For parsing, both forms are accepted, independent of the number of pattern letters.
- **Number:** For formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount. For parsing, the number of pattern letters is ignored unless it's needed to separate two adjacent fields.
- **Year:** The following rules are applied for year:
 - For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits; otherwise it is interpreted as a number.

- For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits. So using the pattern "MM/dd/yyyy", "01/11/12" parses to Jan 11, 12 A.D.
- For parsing with the abbreviated year pattern ("y" or "yy"), the format string must interpret the abbreviated year relative to some century. It does this by adjusting dates to be within 80 years before and 20 years after the current time. For example, using a pattern of "MM/dd/yy", the string "01/11/12" would be interpreted as Jan 11, 2012 while the string "05/04/64" would be interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits, will be parsed into the default century. Any other numeric string, such as a one digit string, a three or more digit string, or a two digit string that isn't all digits (for example, "-1"), is interpreted literally. So "01/02/3" or "01/02/003" are parsed, using the same pattern, as Jan 2, 3 AD. Likewise, "01/02/-3" is parsed as Jan 2, 4 BC.

Otherwise, specific forms are applied. For both formatting and parsing, if the number of pattern letters is 4 or more, a long form is used. Otherwise, short or abbreviated form is used.

- **Month:** If the number of pattern letters is 3 or more, the month is interpreted as text; otherwise, it is interpreted as a number.
- **General time zone:** Time zones are interpreted as text if they have names. For time zones representing a GMT offset value, a number is used.
- **RFC 822 time zone:** For formatting, the RFC 822 4-digit time zone format is used.

Examples

The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone.

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700

Appendix D – Writing a User Function

This Appendix describes how to write a TQL User Function. A User Function, identified in a TQL query by a name which begins with a percent sign (“%”), is a user-developed extension to TQL. A User Function is created by writing a Java class that implements the `TQLFunction` interface class. This class, packaged in a jar file and placed in the classpath, e.g., `lib\modules`, is recognized by TQL on start-up and can subsequently be used in *expressions* in the same way as a TQL Function.

The `TQLFunction` class defines the following required methods:

```
String getName()
```

Returns the base name of this user function (without the initial "%").

```
boolean isAggregator()
```

Returns whether the function returns one value for each `extractValues` argument object or one value for all the arguments (an aggregator). The value of this method is typically false. Some functions, such as `COUNT`, are aggregators and must be handled as 'independent' export attributes. In this case, they must return true to `isAggregator()` and the length of the returned array will be one, not the size of the argument array.

```
boolean forceFullSearch(String op, String value)
```

Returns whether this function as used in a *selector* with the operator `op` and argument `value` should search all objects in the TQL *context*. This is typically false and search optimizers/accelerators can be used. The `COUNT` function, on the other hand, must search all concepts when a value of zero is acceptable: `COUNT(MyProp) < 2`. In this case, `forceFullSearch` must return true to inhibit optimization. Returning true generally has negative performance implications.

```
String validateArgument(TQLField field)
```

Returns the empty string if the specified function argument is valid, otherwise returns an error message. Called during query parse. `TQLField` is an object that encapsulates all TQL Attribute forms: TQL Variables, User Variables, and Attribute references.

```
String[] extractValues(Object[] objs, TQLField field)
```

Returns the result(s) of the function as applied to the argument objects. The `field` argument can be used to determine the type of object used in the invocation if multiple types are permitted, e.g. an inverse association vs. a regular association. `objs` is the array of objects resulting from evaluating the function's argument, and `field` is the object that defines the argument type. Returns a String array of function values, this array can be of different size than `objs.length`.

To simplify coding of User Functions, TQL includes the `TQLFunctionAdapter` class that provides default implementations of the `TQLFunction` methods. User Function classes can extend this class to get default behaviors. At a minimum, the `getName` and `extractValues` methods must be overridden. A helper method is included for extracting the default string value of an attribute.

```
public String getName()
```

Returns the empty string. Must be overridden.

```
public boolean isAggregator()
```

Returns false.

```
public boolean forceFullSearch(String op, String value)
```

Returns false.

```
public String validateArgument(TQLField field) {
```

Returns success: the empty string.

```
public String[] extractValues(Object[] objs, TQLField field)
```

Returns an array containing the result of `extractValue` for each of the arguments. The `field` argument is not used. Must be overridden.

```
protected String extractValue(Object obj, TQLField field) {
```

Returns the default string value of a DTS Attribute, e.g. name of a Concept, the name of the target Concept for a ConceptAssociation.

For additional information on the `TQLFunction`, `TQLFunctionAdapter`, `TQLField` and utility `QualifiedObject` classes, see the TQL Javadoc file in the *DTSInstall\bin\tqleditor* folder.

The listing below shows the code for the TQL User Function `%REVERSE(arg)`. This function accepts a `DTSPROPERTY` as an argument and returns, via the `extractValues` method, the Property values with their characters reversed.

```
import com.apelon.dts.client.attribute.DTSPROPERTY;
import com.apelon.dts.client.attribute.DTSPROPERTYType;
import com.apelon.modules.dts.editor.tql.beans.TQLField;

/**
 * Sample TQL User Function that reverses the value of a Property
 * <p>
 * Copyright (c) 2013 Apelon, Inc. All rights reserved.
 * @since 4.0
 */

public class ReverseFunction extends TQLFunctionAdapter {

    public String getName() {
        return "Reverse";
    }

    public String validateArgument(TQLField field) {
        if (field!=null && field.getAttribute()!=null &&
            (field.getAttribute().getObject() instanceof DTSPROPERTYType))
            return "";
        return "Argument of REVERSE function must be a Property Type.";
    }

    public String[] extractValues(Object[] objs, TQLField field) {
        String[] results = new String[objs.length];
        for (int i=0; i<objs.length; i++) {
```

```

        results[i] = (objs[i] instanceof
DTSPProperty)?reverse(((DTSPProperty)objs[i]).getValue()):"";
    }
    return results;
}

//reverse the characters in a string
private String reverse(String s) {
    StringBuffer sb = new StringBuffer();
    for (int i=s.length()-1; i>=0; i--)
        sb.append(s.charAt(i));
    return sb.toString();
}
}

```