NOTE 199 – Table Query Language

Ger van Diepen, ASTRON Dwingeloo

2006 May 27

Abstract

The Table Query Language (TaQL) is an SQL-like high level language to do operations like selection, sort and update on a table. It is a very versatile language with full support for table columns containing array data. It has a very rich set of functions (like cone search and array reduction) making it very suitable for astronomical applications. It also has full support of nested queries. An operation that can be expressed in a single function is the matching of two sky catalogues.
A binding in C++ and Glish is available. A Python binding is expected before fall 2005.

Contents

1 Introduction

The Table Query Language (TaQL, rhymes with bagel (though many people pronounce it as tackle)) makes it possible to select rows from an arbitrary table based on the contents of its columns and keywords. It supports arbitrary complex expressions including extended regular expressions and many functions. TaQL also makes sorting and column selection possible. Furthermore TaQL has commands to create a table and to modify data in a table.

TaQL is modeled after SQL and contains most of SQL's functionality. The most important features of TaQL different from SQL are:

- The result of a SELECT is not ASCII output. Instead it always creates another table (either temporary or persistent). Usually this is a so-called reference table, but if the select column list contains expressions, it is a plain table. A reference table is a table that can be used as any other table, but it does not contain data. Instead it contains references to the rows and columns in the original table. Thus modifying data in a reference table means that effectively the data in the original table are modified.
- Any operand can be a scalar or an N-dimensional array. Many reduction functions can be applied to arrays.
- A very rich set of mathematical functions.
- Support of various types of patterns/regular expressions.
- Specific operators and functions for cone searching (i.e. spatial searching with a search radius).
- An advanced way of specifying intervals.
- No support of indices, thus always a linear search is done. This is, however, very fast, even for very large tables.
- Joins are not supported yet.
- In the SELECT command GROUPBY and HAVING are not supported yet.
- The CALC command exists to calculate an arbitrary expression (including subqueries) on a table. This can be useful to derive values

from a table (e.g. the number of flags set in a measurement set). It can even be used as a desk calculator.

The first sections of this document explain the syntax and show the options. The last sections show the interface to TaQL using Glish or C++. The Glish interface makes it possible to embed Glish variables and expressions in a TaQL command.

2 TaQL Commands

TaQL contains six commands. In the commands shown below the square brackets are not part of the syntax, but indicate the optional parts of the commands.

- selection

  ```
  SELECT [[DISTINCT] column_list] [INTO table [AS type]]
    FROM table_list [WHERE expression]
    [ORDERBY [DISTINCT] sort_list]
    [LIMIT expression] [OFFSET expression] [GIVING table|set]
  ```

  It can be used to get an optionally sorted subset from a table.

- updating

  ```
  UPDATE table_list SET update_list [FROM table_list]
    [WHERE ...] [ORDERBY ...] [LIMIT ...] [OFFSET ...]
  ```

  It can be used to update data in (a subset of) the first table in the table list.

- addition

  ```
  INSERT INTO table_list [(column_list)] VALUES (expr_list)
  ```
  or
  ```
  INSERT INTO table_list [(column_list)] SELECT_command
  ```

  It can be used to add and fill new rows in the first table in the table list.

- deletion

```
DELETE FROM table_list
  [WHERE ...] [ORDERBY ...] [LIMIT ...] [OFFSET ...]
```

It can be used to delete some or all rows from the first table in the table list.

- calculation

```
CALC [FROM table_list CALC] expression
```

It can be used to calculate an expression, in which columns in a table can be used.

- table creation

```
CREATE TABLE table [column_spec] [DMINFO datamanager_list]
```

It can be used to create a new table with the given columns. in a table can be used.

The commands and verbs in the commands are case-insensitive, but case is important in string values and in names of columns and keywords. Whitespace (blanks and tabs) can be used at will.
A later section (Modifying a table) explains the UPDATE, INSERT, and DELETE command in more detail. The CREATE TABLE is explained in section (Creating a table).
Another section (Calculations on a table) explains the CALC command in more detail.

3 Selection from a table

The SELECT is the main TaQL command. It can be used to select a subset of rows and/or columns from a table or to generate new columns based on expressions.

As explained above, the result of a selection is usually a reference table. This table can be used as any other table, thus it is possible to do another selection on it or to update it (which updates the underlying original table). It is, however, not possible to insert rows in a reference table or to delete rows from it.

If the select column list contains expressions, it is not possible to generate a reference table. Instead a normal plain table is generated (which can take some time if it contains large data arrays). It should be clear that updating such a table does not update the original table.

The various parts of the SELECT command are explained in the following sections.
Although the clauses column_list, WHERE, ORDERBY, LIMIT, and OFFSET are optional, at least one of them has to be used. Otherwise no operation is performed on the primary table (which makes no sense). Note that the GIVING clause with a value set is seen as an operation as well.

3.1 SELECT column_list

Columns to be selected can be given as a comma-separated list with names of columns that have to be selected from the primary table in the table_list (see below). If * or no column_list is given, all columns will be selected. It results in a so-called reference table. Optionally a selected column can be given another name in the reference table using `AS name` (where AS is optional). For example:

```
select TIME,ANTENNA1,ANTENNA2,DATA from 3C343.MS
select TIME,ANTENNA1,ANTENNA2,MODEL_DATA AS DATA from 3C343.MS
```

It is possible to change the data type of a column by specifying a data type (see below) after the new column name. Giving a data type (even if the same as the existing one) counts as an expression, thus results in the generation of a plain table. For example:

```
select MODEL_DATA AS DATA FCOMPLEX from 3C343.MS
```

It is possible to use expressions in the column list to create new columns based on the contents of other columns. When doing this, the resulting table is a normal table (because a reference table cannot contain expressions). The new column can be given a name by giving `AS name` after the expression (where AS is optional). If no name is given, a unique name like `Col_1` is constructed. After the name a data type string can be given for the new column. If no data type is given, the expression data type is used.

```
select max(ANTENNA1,ANTENNA2) AS ANTENNA INT from 3C343
select means(DATA,1) from 3C343
```

Note that for subqueries the GIVING clause offers a better (faster) way of specifying a result expression. It also makes it possible to use intervals.

If a column_list is given and if all columns are scalars, the column_list can be preceeded by the word DISTINCT. It means that the result is

made unique by removing the rows with duplicate values in the columns of the column_list. Instead of DISTINCT the synonym NODUPLICATES or UNIQUE can also be used. To find duplicate values, some temporary sorting is done, but the original order of the remaining rows is not changed. Note that support of this keyword is mainly done for SQL compliance. The same (and more) can be achieved with the DISTINCT keyword in the ORDERBY clause with the difference that ORDERBY DISTINCT will change the order.
For full SQL compliance it is also possible to give the keyword ALL which is the opposite of DISTINCT, thus all values are returned. This is the default. Because there is an ambiguity between the keyword ALL and function ALL, the first element of the column list cannot be an expression if the keyword ALL is used.

3.2 INTO table [AS type]

This indicates that the ultimate result of the SELECT command should be written to a table (with the given name). This table can be a reference table, a plain table, or a memory table.

The type argument is optional and can be one of several values:

MEMORY to store the result in a memory table.

PLAIN to store the result in a plain table. Its endian format is determined by the aipsrc definition.

PLAIN_BIG to store the result in a plain table in big-endian format.

PLAIN_LITTLE to store the result in a plain table in little-endian format.

PLAIN_LOCAL to store the result in a plain table in native endian format.

If type is not given, the result will be written in a reference table. However, if expressions are given in the column list of a projection, the result is written in a plain table.
The standard TaQL way to define the output table is the GIVING clause. INTO is available for SQL compliance.

3.3 FROM table_list

The FROM part defines which tables are used in the selection. It is a comma-separated list of table names which can contain path specification and environment variables or the UNIX ~ notation. The first table in the list is the primary table and is used for all columns in the other clauses. Usually only one table is used in which case the list consists of only one table name. E.g.

```
SELECT col1,col2 FROM mytable
    WHERE col1>col2
    ORDERBY col1
```

In this example columns `col1` and `col2` are taken from `mytable`.

However, it is possible to specify more tables in the table_list. In the WHERE clause these secondary tables can be used to take keywords from. E.g.

```
SELECT FROM mytable,othertable
    WHERE col1>othertable.key
```

As shown in the example above a qualifying name (`othertable.`) can be used in the WHERE clause to specify from which table a keyword has to be taken. If no qualifying name is given, the keyword (or column) is taken from the primary table (i.e., the first table in the table_list). This means that qualifying names are only needed in special cases. The qualifying name can not contain special characters like a slash. Therefore a table_name needs an explicit shorthand alias if it contains special characters.

The full table_list syntax is:
`table_name1 [shorthand1], table_name2 [shorthand2], etc.`
The shorthand defaults to the table_name. A shorthand is needed if the table name contains non-alphanumeric characters. In the following example shorthand `my` is not really needed. Shorthand `other` is needed though.

```
SELECT FROM mytable my, ~user/othertable other
    WHERE my.col1>other.key
```

Similar to SQL and OQL the shorthand can also be given using `AS` or `IN`. E.g.

```
SELECT FROM mytable AS my, other IN ~user/othertable
```

Note that if using `IN`, the shorthand has to preceed the table name. It can be seen as an iterator variable.

There are three special ways to specify a table:

1. A table name can be taken from a keyword in a previously specified table. This can be useful in a subquery. The syntax for this is the same as that for specifying keywords in an expression. E.g.

   ```
   SELECT FROM mytable tab
       WHERE col1 IN [SELECT subcol FROM tab.col2::key]
   ```

   In this example `key` is a table keyword of column `col2` in table `mytable` (note that `tab` is the shorthand for `mytable` and could be left out).
   It can also be used for another table in the main query. E.g.

   ```
   SELECT FROM mytable, ::key subtab
       WHERE col1 > subtab.key1
   ```

   In this example the keyword `key1` is taken from the subtable given by the table keyword `key` in the main table.
   If a keyword is used as the table name, the keyword is searched in one of the tables previously given. The search starts at the current query level and proceeds outwards (i.e., up to the main query level). If a shorthand is given, only tables with that shorthand are taken into account. If no shorthand is given, only primary tables are taken into account.

2. Like in OQL it is possible to use a nested query command in the FROM clause. This is a normal query command enclosed in square brackets or parentheses. It results in a temporary table which can thereafter be used as a table in the rest of the query command. A shorthand has to be defined for it in order to be able to refer to that table.
   Use of a query in the FROM command can be useful to avoid multiple equal subqueries. E.g.

   ```
     select from MS,
      [select from MS where sumsqr(UVW[1:2]) < 625]
       as TIMESEL
      where TIME in [select distinct TIME from TIMESEL]
       &&  any([ANTENNA1,ANTENNA2] in
         [select from TIMESEL giving
   [iif(UVW[3] < 0, ANTENNA1, ANTENNA2)]])
   ```

   is a command to find shadowed antennas for the VLA. Without the query in the FROM command the subqueries in the remainder of the

command would have been more complex. Furthermore, it would have been necessary to execute that select twice.
The command is quite complex and cannot be fully understood before reading the rest of this note. Note, however, that the command uses the shorthand `TIMESEL` to be able to use the temporary table in the subqueries.

3. Normally only persistent tables (i.e. tables on disk) can be used. However, it is also possible to use transient tables in a TaQL command given in Glish or C++. This is done by passing one or more table objects to the function executing the TaQL command. In the TaQL command a $-sign followed by a sequence number has to be given to indicate the correct object containing the transient table. E.g. if two table objects are passed $1 indicates the first table, while $2 indicates the second one.

3.4 WHERE expression

It defines the selection expression which must have a boolean scalar result. A row in the primary table is selected if the expression is true for the values in that row. The syntax of the expression is explained in a later section.

3.5 ORDERBY sort_list

It defines the order in which the result of the selection has to be sorted. The sort_list is a comma separated list of expressions.
The sort_list can be preceeded by the word `ASC` or `DESC` indicating if the given expressions are by default sorted in ascending or descending order (default is ASC). Each expression in the sort_list can optionally be followed by `ASC` or `DESC` to override the default order for that particular sort key.
To be compliant with SQL whitespace can be used between the words ORDER and BY.

The word ORDERBY can optionally be followed by DISTINCT which means that only the first row of multiple rows with equal sort keys is kept in the result. To be compliant with SQL dialects the word UNIQUE or NODUPLICATES can be used instead of DISTINCT.

An expression can be a scalar column or a single element from an array column. In these cases some optimization is performed by reading the entire column directly.
It can also be an arbitrarily complex expression with exactly the same syntax rules as the expressions in the WHERE clause. The resulting data type of

the expression must be a standard scalar one, thus it cannot be a Regex or DateTime (see below for a discussion of the available data types). E.g.

```
ORDERBY col1, col2, col3
ORDERBY DESC col1, col2 ASC, col3
ORDERBY NODUPLICATES uvw[1] DESC
ORDERBY square(uvw[1]) + square(uvw[2])
ORDERBY datetime(col)      incorrect data type
ORDERBY mjd(datetime(col)) is correct
```

3.6 LIMIT expression

It indicates that up to N rows have to be selected, where N is the result of the expression.
`LIMIT` is applied after `ORDERBY`, so it is particularly useful in combination with `ORDERBY` to select, for example, the highest 10 values.

3.7 OFFSET expression

It indicates that the first N rows of the selection have to be skipped. As `LIMIT` it is particularly useful in combination with `ORDERBY`.

3.8 GIVING table|set

It indicates that the ultimate result of the SELECT command should be written to a table (with the given name). One can also give an option using `AS option`.
Another (more SQL compliant) way to define the output table is the INTO clause. See INTO for a more detailed description including the possible options.

It is also possible to specify a set in the GIVING clause instead of a table name. This is very useful if the result of a subquery is used in the main query. Such a set can contain multiple elements Each element can be a single value, range and/or interval as long as all elements have the same data type. The parts of each element have to be expressions resulting in a scalar.

In the main query and in a query in the FROM command the GIVING clause can only result in a table and not in a set.
To be compliant with SQL dialects, the word SAVETO can be used instead of GIVING. Whitespace can be given between SAVE and TO.

4 Expressions

An arbitrary expression can be used in the WHERE clause, as a sort key in the ORDERBY clause, in an element in the set in the GIVING clause, or as a value in the INSERT and UPDATE command. Note that the expression result must be a boolean scalar if used in the WHERE clause. If used in ORDERBY the result can also be a numeric or string scalar. If used in GIVING it can also be a DateTime scalar. If used as a new value in the UPDATE or INSERT command it can also be an array.
The expression in the clause can be as complex as one likes using the standard arithmetic, comparison, and logical operators. Parentheses can be used to group subexpressions.
The operands in an expression can be table columns, table keywords, constants, functions, sets and intervals, and subqueries.
The index operator can be used to take a single element or a subsection from an array expression.
E.g.

```
column1 > 10
column1 + arraycolumn[index] >= min (column2, column3)
column1 IN [expr1 =:< expr2]
```

The last example shows a set with a continuous interval.

4.1 Data Types

Internally TaQL uses the following data types:

Bool

Double which includes integers and times/positions

Complex which includes single and double precision complex

String Operator + can be used (concatenation).

Regex which is formed by the functions `regex`, `pattern`, and `sqlpattern` or by a special constant (see discussion below).

DateTime which represents a date/time. There are several functions acting on a date/time. Also operator + and - can be used.

Scalars and arbitrarily shaped arrays of these data types can be used. However, arrays of Regex are not possible.
If an operand or function argument with a non-matching data type is used, TaQL can do the following automatic conversions:
- from Double to Complex
- from String to DateTime

Externally TaQL supports any possible data type of a table column or keyword. In some commands (column list and CREATE TABLE) it is possible to specify the data type of a column. There are several synonyms. The possible types and values are:

B		BOOL	BOOLEAN
U1		UCHAR	BYTE
I2		SHORT	SMALLINT
U2	UI2	USHORT	USMALLINT
I4		INT	INTEGER
U4	UI4	UINT	UINTEGER
R4	FLT	FLOAT	
R8	DBL	DOUBLE	
C4	FC	FCOMPLEX	COMPLEX
C8	DC	DCOMPLEX	
S		STRING	

4.2 Constants

Scalar constants of the various data types can be formed as follows (which is very similar to Glish):

- A Bool constant is the value `T` or `F`.
- A Double constant can be any integer or floating-point number.
  Another way to define a Double constant is by means of a Time or Position. Such a constant is always converted to radians. It can be given in several ways:
  - An integer or floating-point number immediately followed by a unit (thus without whitespace). E.g. `12.43deg`
    Some valid units are deg, arcmin, arcsec (or as), rad. The units can be scaled by preceeding them with a letter (e.g. mrad is millirad).
  - A time/position in hours in HMS format. E.g. `12h34m34.5` or `8h`

- A position in degrees in DMS format. E.g. `12d34m34.5` or `8d`
- A position in degrees in dot format. Note that the dots for degrees and minutes must always be present. E.g. `12.34.34.5` or `8..34.5`

- The imaginary part of a Complex constant is formed by a Double constant immediately followed by a lowercase **i**. A full Complex constant is formed by adding another Double constant as the real part. E.g.

```
1.5 + 2i
2i+1.5            is identical
```

  Note that a full Complex constant has to be enclosed in parentheses if, say, a multiplication is performed on it. E.g.

```
2 * (1.5+2i)
```

- A String constant has to be enclosed in " or ' and can be concatenated (as in C++). E.g.

```
"this is a string constant"
'this is a string constant containing a "'
"ab'cd"'ef"gh'
    which results in constant   ab'cdef"gh
```

- A Regex constant can be given in three flavours:

  - as a UNIX shell (file name) style pattern.
    `p/pattern/` can be used after operator `~` and `!~`. Instead of a slash, the characters % and # can also be used as delimiter, as long as the same delimiter is used on both sides. The delimiter can not be part of the pattern itself.
    Function `pattern` can be used after operator = and !=.
    A pattern uses the special character ? for a single arbitrary character and * for zero or more arbitrary characters. Furthermore square brackets can be used for a choice from a set of single characters and braces can be used for a choice from a set of pattern parts.

```
name~p/3c*/
col=pattern('[_A-Za-z]*')
name !~ p#/usr/*.{h,cc}#
```

- as an SQL style pattern.
  A string constant giving the pattern can be used after operator `LIKE`.
  Function `sqlpattern` can be used after operator = and !=.
  An SQL style pattern uses the special character _ for a single arbitrary character and % for zero or more arbitrary characters.

  ```
  name LIKE '3c%'
  ```

- as an extended regular expression.
  `m/pattern/` can be used after operator `~` and `!~`. A slash, % and # can be used as delimiter.
  Function `pattern` can be used after operator = and !=. It uses many special characters. See class Regex for a full description.

  ```
  name ~ m/[a-zA-Z][a-zA-Z0-9]*/
  name = regex('[a-zA-Z][a-zA-Z0-9]')
  ```

Note that the functions and operator LIKE work on any string, thus they can be used with any string expression.

- A DateTime constant can be formed in 2 ways:

  1. From a String constant using the `datetime` function. In this way all possible formats as explained in class `MVTime` are supported. E.g.

     ```
     datetime ("11-Dec-1972")
     ```

  2. A more convenient way is to specify it directly. Since this makes use of the delimiters - or /, it conflicts with the expression grammar as such. However, such conflicts can be solved by using whitespace in a expression and it is believed that in practice the convenience surpasses the possible conflicts.
     A large subset of the MVTime formats is supported. A DateTime has to be specified as `date/time` or `date-time`, where the time

part (including the / or - delimiter) is optional. The possible date formats are:
- YYYY/MM/DD or DD-MM-YYYY
- DD-MMMMMMMM-YY where the - is optional and MMM-MMMM is the case-insensitive name of the month (at least 3 letters).
- YYYY//DDD or DDD–YYYY where DDD is the day number in the year.
In the DMY format, 2000 is added if year<50 and 1900 is added if 50<=year<100.
If MM>12, YYYY will be incremented accordingly.
The general time format in a DateTime constant is:
- hh:mm:ss.s
where the delimiter **h** or **H** can be used for the first colon and **m** or **M** for the second. Trailing parts can be omitted. E.g.

```
10-2-97
10-02-1997
10-February-97
10feb97
1997/2/10          are all identical

1May96/3:          : (or h) is mandatory
1May96/3:0
1May96/3:0:0
1May96/3h          h (or :) is mandatory
1May96/3H0
1May96/3h0M
1May96/3hm0.0
```

A DateTime constant with the current date/time can be made by using the function `datetime` without arguments.

Constant N-dimensional arrays cannot be formed directly. It is possible though to form a constant bounded discrete set. If needed such a set is automatically transformed to a 1-dimensional array. E.g.

```
[1:10]
```

is equal to a 1D array of 10 elements with values 1..10.

Furthermore it is possible to use the `array` function to transform an array to another shape. This function is very similar to the same function in glish. E.g.

```
array([1:10],10,4)
```

creates an array with shape [10,4]. Each row contains the values 1..10. Please note that in TaQL arrays are stored in Fortran order, thus the first index varies most rapidly (which is opposite to C).

Similar to glish the value array is wrapped if the created array has more elements. Unlike glish the shape does not need to be defined by scalars, but it can also be defined by another set. Thus:

```
array(F,shape(DATA))
```

is a boolean array filled with F with the same shape as the DATA array.

4.3 Operators

The following operators can be used (with their normal meaning and precedence) for scalars:

- Unary + and `-`
  Can only be used with double and complex operands. They do not have a higher precedence than the binary operators.
  E.g. `-3^2` results in `-9`.

- Unary `!` (or **NOT**)
  Logical NOT operator. Can only be used with Bool operands.

- Binary ^, *, /, %, +, and `-`
  % is the modulo operator. E.g. `3%1.4` results in `0.2` and `-10%3` results in `-1`.
  ^ is the power operator. Note that because of the precedence rules `-3^2` results in `-9`.
  All operators are left-associative, except ^ which is right-associative; thus `2^1^2` results in `2`.
  Operator % can only be used for double operands, while the others can be used for double and complex operands. Operator + can also be used for:
  - Addition (i.e., concatenation) of 2 strings.
  - Addition of a double (unit days) and a DateTime (resulting in a DateTime).

Operator - can also be used for:
- Subtraction of a double (unit days) from a DateTime (resulting in a DateTime).
- Subtraction of 2 DateTimes (resulting in a double with unit days).

- **==, !=, >, >=, <, and <=**
Can be used with any operand as long as their data types conform. Operator >, >=, <, and <= cannot be used for Regex. They use the norm for Complex values.
To be compliant with SQL = can be used for == and <> can be used for !=.

- `~`, `!~`, and **LIKE** and **NOT LIKE**.
`~` and `!~` have to be used with regular expression constants.
LIKE exists for SQL compliancy. It operates on a string giving a pattern with special characters % (0 or more characters) and _ (any single character).

- **EXISTS** and **NOT EXISTS**
Can be used to test if a subquery finds at least N matching rows. The value for N is taken from the LIMIT clause; if LIMIT is not given it defaults to 1. The subquery loop stops as soon as N matching rows are found. E.g.

```
EXISTS(select from ::ANTENNA where NAME=''somename'' LIMIT 2)
```

results in T if at least 2 matching rows in the ANTENNA table were found.

- **IN** and **NOT IN**
Can be used to test if a value is an element in a set of values, ranges, and/or intervals. (See the discussion of sets).

- **BETWEEN** and **NOT BETWEEN**
Is a special form of IN with a single closed interval. It exists for SQL compliance. It can be used to test if a value is between two other values. E.g.
`col BETWEEN 1 AND 3` and `col in [1=:=3]` are the same.

- **INCONE** and **NOT INCONE**
Can be used to do a cone search. (See the discussion of cone search functions).

- **&&** and ||
  Logical AND and OR operator.
  To be compliant with SQL the words `AND` and `OR` can be used instead of the symbols. These words are case insensitive.
  Also **&** and | can be used.
  These operators can only be used with Bool operands.

The precedence order is:

```
^
unary +, -, !
*, /, %
+, -
==, !=, >, >=, <, <=, IN, INCONE, BETWEEN, EXISTS, LIKE, ~,
!~
&&
||
```

All these operators can be used in the same way for arrays (also a mix of scalar and array).

4.4 Functions

Some functions have 2 names. One name is the AIPS++/Glish name, while the other is the name as used in SQL. In the following tables the function names are shown in uppercase, while the result and argument types are shown in lowercase. Note, however, that function names are case-insensitive.

Sets, and in particular subqueries, can result in a 1-dim array. This means that the functions accepting an array argument can also be used on a set or the result of a subquery. E.g.

```
WHERE datecol IN date([11-Nov-97,16-Dec-97,14-Jan-98])
```

4.4.1 String functions

These functions can be used on a scalar or an array argument.

`double STRLENGTH(string), double LEN(string)`
: Returns the number of characters in a string (trailing whitespace is significant).

`string UPCASE(string), string UPPER(string)`
: Convert to uppercase.

`string DOWNCASE(string), string LOWER(string)`
: Convert to lowercase.

`string TRIM(string)`
: Removes trailing whitespace.

4.4.2 Regex functions

The syntax for forming a regex and pattern are explained in class Regex. These functions can only be used on a scalar argument.

`regex REGEX(string)`
: Handle the given string as a regular expression.

`regex PATTERN(string)`
: Handle the given string as a UNIX filename-like pattern and convert it to a regular expression.

`regex SQLPATTERN(string)`
: Handle the given string as an SQL-style pattern and convert it to a regular expression. An SQL-style pattern has two special character:
% for zero or more arbitrary characters
_ for one arbitrary character

A regex formed this way can only be used in a comparison == or !=. E.g.
`upcase(object) == pattern('3C*')`
to find all 3C objects in a catalogue. See also the discussion on constants for other ways of forming and testing regular expressions.

4.4.3 Date/time functions

These functions can be used on a scalar or an array argument.

`DateTime DATETIME(string), DateTime CTOD(string)`
: Parse the string and convert it to a date/time. The syntax of date/time is explained in class MVTime.

`DateTime MJDTODATE(double)`
: The double value, which has to be a MJD (ModifiedJulianDate), is converted to a DateTime.

`DateTime DATE(DateTime)`
: Get the date (i.e., remove the time part). This function is needed in
`DATE(column) == 12Feb1997`
if the column contains date/times with times>0.

`double MJD(DateTime)`
: Get the DateTime as a MJD (ModifiedJulianDate).

`double YEAR(DateTime)`
: Get the year (which includes the century).

`double MONTH(DateTime)`
: Get the month number (1-12).

`double DAY(DateTime)`
: Get the day number (1-31).

`double WEEKDAY(DateTime), double DOW(DateTime)`
: Get the weekday number (1=Monday, ..., 7=Sunday).

`string CMONTH(DateTime)`
: Get the name of the month (Jan ... Dec).

`string CWEEKDAY(DateTime), string CDOW(DateTime)`
: Get the name of the weekday (Mon .. Sun).

`double WEEK(DateTime)`
: Get the week number in the year (1 ... 53).
Note that week 1 is the week containing Jan 4th.

`double TIME(DateTime)`
: Get the time part of the day. It is converted to radians to be compatible with the internal representation of times/positions. In that way the function can easily be used as in:
`TIME(date) > 12h`

All functions can be used without an argument in which case the current date/time is used. E.g. `DATE()` results in the current date.
It is possible to give a string argument instead of a date. In this case the string is parsed and converted to a date (i.e., in fact the function DATETIME is used implicitly).

4.4.4 Comparison functions

Two functions make it possible to compare 2 double or complex values within a tolerance. They can be used on scalar and array arguments (and a mix of them). The tolerance must be a scalar though.

`bool NEAR(numeric val1, numeric val2, double tol)`
: Tests in a relative way if a value is near another. Relative means that the magnitude of the numbers is taken into account.
It returns `abs(val2 - val1)/max(abs(val1),abs(val2)) < tol`.
If `tol<=0`, it returns `val1==val2`. If either val is 0.0, it takes care of area around the minimum number that can be represented. The default tolerance is 1.0e-13.

`bool NEARABS(numeric val1, numeric val2, double tol)`
: Tests in an absolute way if a value is near another. Absolute means that the magnitude of the numbers is not taken into account.
It returns `abs(val2 - val1) < tol`. The default tolerance is 1.0e-13.

`bool ISNAN(numeric val)`
: Tests if a numeric value is a NaN (not-a-number).

4.4.5 Mathematical functions

Several functions can operate on double or complex arguments. The data types of such functions is given as 'numeric'. They can be used on scalar and array arguments (and a mix of them).

`double PI()`
: Returns the value of **pi**.

`double E()`
: Returns the value of **e** (is equal to `EXP(1)`).

`numeric SIN(numeric)`

`numeric SINH(numeric)`

`double ASIN(double)`

`numeric COS(numeric)`

`numeric COSH(numeric)`

`double ACOS(double)`

`double TAN(double)`

`double TANH(double)`

`double ATAN(double)`

`double ATAN2(double y, double x)`
Returns `ATAN(y/x)` in correct quadrant.

`numeric EXP(numeric)`

`numeric LOG(numeric)`
Natural logarithm.

`numeric LOG10(numeric)`

`numeric POW(numeric, numeric)`
The same as operator ^.

`numeric SQUARE(numeric), numeric SQR(numeric)`
The same as ^2, but much faster.

`numeric SQRT(numeric)`

`complex COMPLEX(double, double)`

`numeric CONJ(numeric)`

`double REAL(numeric)`
Real part of a complex number.

`double IMAG(numeric)`
Imaginary part of a complex number.

`double NORM(numeric)`

`double ABS(numeric), double AMPLITUDE(numeric)`

`double ARG(numeric), double PHASE(numeric)`

`numeric MIN(numeric, numeric)`

`numeric MAX(numeric, numeric)`

`double SIGN(double)`
Returns -1 for a negative value, 0 for zero, 1 for a positive value.

`double ROUND(double)`
: Rounds the absolute value of the number. E.g. `ROUND(-1.6) = -2`.

`double FLOOR(double)`
: Works towards negative infinity. E.g. `FLOOR(-1.2) = -2`

`double CEIL(double)`
: Works towards positive infinity.

`double FMOD(double, double)`
: The same as operator %.

Note that the trigonometric functions need their arguments in radians.

4.4.6 Array to scalar reduction functions

The following functions reduce an array to a scalar. They are meant for an array, but can also be used for a scalar.

`bool ANY(bool)`
: Is any element true?

`bool ALL(bool)`
: Are all elements true?

`double NTRUE(bool)`
: Number of true elements.

`double NFALSE(bool)`
: Number of false elements.

`numeric SUM(numeric)`
: Return sum of all elements.

`numeric SUMSQUARE(numeric), numeric SUMSQR(numeric)`
: Return sum of all squared elements.

`numeric PRODUCT(numeric)`
: Return product of all array elements.

`double MIN(double)`
: Return minimum of all array elements.

`double MAX(double)`
: Return maximum of all array elements.

`double MEAN(double), double AVG(double)`
Return mean of all array elements.

`double VARIANCE(double)`
Return variance (the sum of
`(a(i) - mean(a))**2/(nelements(a) - 1)`.

`double STDDEV(double)`
Return standard deviation (the square root of the variance).

`double AVDEV(double)`
Return average deviation. (the sum of
`abs(a(i) - mean(a))/nelements(a)`.

`double MEDIAN(double)`
Return median (the middle element). If the array has an even number of elements, the mean of the two middle elements is returned.

`double FRACTILE(double, doublescalar fraction)`
Return the value of the element at the given fraction. Fraction 0.5 is the same as the median.

4.4.7 Array to array reduction functions

These functions reduce an array to a smaller array by collapsing the given axes using the given function. The axes are the last argument(s). They can be given in two ways:
- As a single set argument; for example, `maxs(ARRAY,[1:2])`
- As individual scalar arguments; for example, `maxs(ARRAY,1,2)`
For example, using `MINS(array,1,2)` for a 3-dim array results in a 1-dim array where each value is the minimum of each plane in the cube.
Axes numbers are 1-relative, thus 1 is the first axis.
Axes numbers exceeding the dimensionality of the array are ignored. Thus, for example, `maxs(ARRAY,[2:10])` works for arrays of virtually any dimensionality and results in a 1-dim array.
The function names are the 'plural' forms of the functions in the previous table. They can only be used for arrays, thus not for scalars.

`bool ANYS(bool)`
Is any element true?

`bool ALLS(bool)`
Are all elements true?

`double NTRUES(bool)`
Number of true elements.

`double NFALSES(bool)`
Number of false elements.

`numeric SUMS(numeric)`
Return sum of elements.

`numeric SUMSQUARES(numeric), numeric SUMSQRS(numeric)`
Return sum of squared elements.

`numeric PRODUCTS(numeric)`
Return product of elements.

`double MINS(double)`
Return minimum of elements.

`double MAXS(double)`
Return maximum of elements.

`double MEANS(double), double AVG(double)`
Return mean of elements.

`double VARIANCES(double)`
Return variance (the sum of `(a(i) - mean(a))**2/(nelements(a) - 1)`.

`double STDDEVS(double)`
Return standard deviation (the square root of the variance).

`double AVDEVS(double)`
Return average deviation. (the sum of `abs(a(i) - mean(a))/nelements(a)`.

`double MEDIANS(double)`
Return median (the middle element). If the array has an even number of elements, the mean of the two middle elements is returned.

`double FRACTILES(double, doublescalar fraction)`
Return the value of the element at the given fraction. Fraction 0.5 is the same as the median.

4.4.8 Miscellaneous functions

`bool ISDEFINED(anytype)`
: Return false if the value in the current row is undefined. Is useful to test if a cell in a column with variable shaped arrays contains an array. It can also be used to test if a field in a record is defined.

`double NELEMENTS(anytype), double COUNT(anytype)`
: Return number of elements in an array (1 for a scalar).

`double NDIM(anytype)`
: Return dimensionality of an array (0 for a scalar).

`doublearray SHAPE(anytype)`
: Return shape of an array (returns an empty array for a scalar).

`double ROWNUMBER()`
: Return the row number being tested (first row is row number 1).
In combination with function RAND it can, for instance, be used to select arbitrary rows from a table.

`double ROWID()`
: Return the row number in the original table. This is especially useful for returning the result of a selection of a subtable of an AIPS++ measurement set (see also sections Subqueries and Examples).

`double RAND()`
: Return (per table row) a uniformly distributed random number between 0 and 1 using a Multiplicative Linear Congruential Generator. The seeds for the generator are deduced from the current date and time, so the results are different from run to run.
The function can, for instance, be used to select a random subset from a table.

`anytype IIF(cond,arg1,arg2)`
: This is a special funtion which operates like the ternary `?:` operator in C++. If all arguments are scalars, the result is a scalar, otherwise an array. In the latter case possible scalar arguments are virtually expanded to arrays. IIF evaluates the condition for each element. If True, it takes the corresponding element of `arg1`, otherwise of `arg2`.

`anytypearray ARRAY(anytype,shape)`
: This function creates an array of the given type and shape. The shape

is given in the last argument(s). It can be given in two ways:
- As a single set argument; for example, `array(0,[3,4])`
- As individual scalar arguments; for example, `array(0,3,4)`
The first argument gives the values the array is filled with. It can be a scalar or an array of any shape. To initialize the created array, the value array is flattened to a 1D array. Its successive values are stored in the created array. If the new array has more values than the value array, the value array is reset to its beginning and the process continues.

4.4.9 Cone search functions

Cone search functions make it possible to test if a source is within a given distance of a given sky position. The expression

```
cos(0d1m) < sin(52deg) * sin(DEC) +
            cos(52deg) * cos(DEC) * cos(3h30m - RA)
```

could be used to test if sources with their sky positions defined in columns `RA` and `DEC` are within 1 arcmin of the given sky position.
The cone search functions implement this expression making life much easier for the user. Because they can also operate on arrays of positions, searching in multiple cones can be done simultaneously. That makes it possible to find matching source positions in two catalogues as shown in an example at the end of this section.

The arguments in all functions are described below. All of them have to be given in radians. However, usually one does not need to bother because TaQL makes it possible to specify positions in many formats which are automatically converted to radians.

`SOURCES`
is a set or array giving the positions of one or more sources (e.g. in equatorial coordinates) to be tested. Normally these are columns in a table. If argument name `SOURCE` is mentioned, only a single source can be used.
For example:
`[RA,DEC]` for scalar columns RA and DEC.
`SKYPOS` for a column SKYPOS containing 2-element vectors with RA and DEC.

`CONES`
is a set or array giving the center positions and radii of one or more

cones (e.g. as RA,DEC,radius). Usually this will be given by the user as constants.
For example:
`[12:13:54, -5.3.34, 0d1m]` for a single cone.
`[12:13:54, -5.3.34, 0d1m, 1:2:3, 4.5.6, 0d1m]` for two cones.

`CONEPOS`
: is a set or array giving the positions of one or more cone centers (e.g. as RA,DEC).

`RADII`
: is a scalar, set or array giving one or more radii. Each radius is applied to all positions in `CONEPOS`. Specifying a cone as `CONEPOS,RADIUS` is easier than specifying it as `CONES` if the same radius has to be used for multiple cones.
For example:
`[12:13:54, -5.3.34, 1:2:3, 4.5.6], 0d1m` is the same as the second `CONES` example above.

The following cone search functions are available.

`bool ANYCONE(SOURCE,CONES)`
: Return `T` if the source is contained in one of the cones. Operator `INCONE` is a synonym. So `ANYCONE(SOURCE,CONES)` is the same as `SOURCE INCONE CONES`.

`bool ANYCONE(SOURCE,CONEPOS,RADII)`
: It does the same as above.

`double FINDCONE(SOURCES,CONES)`
: Return the 1-based index of the first cone containing the source. If a single source is given, the result is a scalar. If multiple sources are given, the result is an array with the same shape as the source array.

`double FINDCONE(SOURCES,CONEPOS,RADII)`
: It does the same as above. Note that in this case each radius is applied to each cone, so the index is a combination of the two (with the radius as the most rapidly varying axis).

`bool CONES(SOURCES,CONES)`
: Return a 2-dim bool array. The length of the first axis is the number of cones. The length of the second axis is number of sources. An element `(i,j)` in the array is `T` if source `j` is contained in cone `i`.

`bool CONES(SOURCES,CONEPOS,RADII)`
: It does the same as above. However, the result is a 3-dim array with the radii as the first axis, cones as second and sources as third.

Please note that `ANYCONE(SOURCE,CONES)` does the same as `any(CONES(SOURCE,CONES))`, but faster.
Function `CONES` makes it possible to do catalogue matching. For example, to find sources matching other sources in the same catalogue (within a radius of 10 arcseconds):

```
CALC FROM table.cat CALC CONES([RA,DEC],
     [SELECT FROM table.cat GIVING [RA,DEC]], 0d0m10)
```

Note that in this example the SELECT clause returns an array with positions which are used as the cone centers. So each source in the catalogue is tested against every source. It makes it an N-square operation, thus potentially very expensive. The result is a 4-dim boolean array with shape `[1,nrow,1,nrow]` which can be processed in Glish. Please note that the `CONES` function results for each row in a array with shape `[1,nrow,1]`.
The query could be done with multiple radii, for example also with 1 arcsecond and 1 arcminute.

```
CALC FROM table.cat CALC CONES([RA,DEC],
     [SELECT FROM table.cat GIVING [RA,DEC]], [0d0m1, 0d0m10, 0d1m])
```

resulting in an array with shape `[3,nrow,1,nrow]`. It would give a better indication how close sources are.

4.5 Table Columns

A column can contain a scalar or an array value. Note that only columns in the primary table can be handled directly. A column in another table can be used via a subquery. E.g.

```
SELECT FROM tab WHERE col >
      mean([SELECT othercol FROM othertab])
```

An expression has to contain at least one column, since columns are the only variable part in it. That is, a row can only be selected or sorted by means of the column values in each row.

The name of a column can contain alphanumeric characters and underscores. It should start with an alphabetic character or underscore. A column name is case-sensitive.

It is possible to use other characters in the name by escaping them with a backslash. E.g. `DATE\-OBS`.
In the same way a numeric character can be used as the first character of the column name. E.g. `\1stDay`.
Because several words are used in the language, they cannot be used directly as column names. The reserved words are:

```
ALL AND AS ASC BETWEEN CREATETABLE DELETE DESC DISTINCT
EXISTS EXCEPT FROM GIVING GROUPBY HAVING
IN INCONE INSERT INTERSECT INTO JOIN LIKE LIMIT
MINUS NODUPLICATES NOT OFFSET ON OR ORDERBY
SAVETO SELECT SET UNION UNIQUE UPDATE VALUES WHERE
```

They can, however, be used as a column name by escaping them with a backslash. E.g. `\IN`.
Note that in C++ and Glish a backslash itself has to be escaped by another backslash. E.g. in Glish: `tab.query('DATE\\-OBS>10MAR1996')`.

If a column contains a record, one has to specify a field in it using the dot operator; e.g. `col.fld` means use field `fld` in the column. It is fully recursive, so `col.fld.subfld` can be used if field `fld` is a record in itself. Alas records in columns are not really supported yet. One can specify fields, but thereafter an error message will be given.

4.6 Table Keywords

It is possible to use table or column keywords, which can have a scalar or an array value. A table keyword has to be specified as `::key`. In an expression the `::` part can be omitted if there is no column with such a name. A column keyword has to be specified as `column::key`.
Note that the `::` syntax is chosen, because it is similar to the scope operator in C++.
As explained in the FROM clause of the syntax section, keywords from the primary table and from secondary tables can be used. If used from a secondary table, it has to be qualified with the (shorthand) name of the table. E.g.
`sh.key` or `sh.::key`
takes table keyword `key` from the table with the shorthand name `sh`.

If a keyword value is a record, it is possible to use a field in it using the dot operator. E.g. `::key.fld` to use field `fld`. It is fully recursive, so if the field is a record in itself, a subfield can be used like `col::key.fld.subfld`

A keyword can be used in any expression. It is evaluated immediately and transformed to a constant value.

4.7 Array Index Operator

It is possible to take a subsection or a single element from an array column, keyword or expression using the index operator `[index1,index2,...]`. The rules for this are similar to those used in Glish. Taking a single element can be done as:

```
array[1]
array[1, some_expression]
```

Taking a subsection can be done as:

```
array[start1:end1:incr1, start2:end2:incr2, ...]
```

If a start value is left out it defaults to the beginning of that axis. An end value defaults to the end of the axis and an increment defaults to one. If an entire axis is left out, it defaults to the entire axis.
E.g. an array with shape [10,15,20] can be subsectioned as:

```
[,,3]                resulting in an array of shape [10,15,1]
[2:4, ::3, 2:15:2]   resulting in an array of shape[3,5,7]
```

The examples show that an index can be a simple constant (as it will usually be). It can also be an expression which can be as complex as one likes. The expression has to result in a positive double value which will be truncated to an integer length. **Note** that as in Glish, array indices start at 1.
For fixed shaped arrays checking if array bounds are exceeded is done at parse time. For variable shaped arrays it can only be done per row. If array bounds are exceeded, an exception is thrown. In the future a special undefined value will be assigned if bounds of variable shaped arrays are exceeded to prevent the selection process from aborting due to the exception.

Note that the index operator will be applied directly to a column. This results in reading only the required part of the array from the table column on disk. It is, however, also possible to apply it to a subexpression (enclosed in parentheses) resulting in an array. E.g.

```
arraycolumn[2,3,4] + 1
(arraycolumn + 1)[2,3,4]
```

can both be used and have the same result. However, the first form is much faster, because only a single element is read (resulting in a scalar) and 1 is added to it. The second form results in reading the entire array. 1 is added to all elements and only then the requested element is taken.
From this example it should be clear that indexing an array expression has to be done with care.

4.8 Sets and intervals

As in SQL the operator `IN` can be used to do a selection based on an array or a set. E.g.

```
SELECT FROM table WHERE column IN expr1
```

In this example `expr1` is the array result of an expression. The result of operator `IN` is true if the column value matches one of the values in the array. It is also possible to use a scalar as the righthand of operator `IN`. So if `expr1` is a scalar, operator `IN` gives the same result as operator `==`.
It is also possible to define a set explicitly. E.g.

```
SELECT FROM table WHERE column IN [expr1, expr2, expr3]
```

This example shows that (in its simplest form) a set consists of one or more values (which can be arbitrary expressions) separated by commas and enclosed in square brackets. The elements in a set have to be scalars and their data types have to be the same. The square brackets can be left out if the set consists of only one element. For SQL compliance parentheses can be used instead of square brackets.
An element in a set can, however, be more complicated and can define multiple values or an interval. The possible forms of a set element are:

1. A single value as shown in the example above.

2. `start:end:incr`. This is similar to the way an array index is specified. Incr defaults to 1. End defaults to an open end (i.e., no upper bound) and results in an unbounded set. Start and end can be a double or a datetime. Incr has to be a double. Some examples:

   ```
   1:10     means 1,2,...,10
   1:10:2   means 1,3,5,7,9
   1::2     means all odd numbers
   1:       means all positive integer numbers
   date('18Aug97')::2   means every other day from 18Aug97 on
   ```

These examples show constants only, but `start`, `end`, and `incr` can be any expression.
Note that :: used here can conflict with the :: in the keywords. E.g. `a::b` is scanned as a keyword specification. If the intention is `start::incr` it should be specified as `a:  :b`. In practice this conflict will hardly ever occur.

3. Continuous intervals can be specified for double, string, and datetime. The specification of an interval resembles the mathematical notation `1<x<5`, where x is replaced by :. An open interval side is indicated by `<`, while a closed interval side is indicated by `=`.
   Another way to specify intervals is using curly and/or angle brackets. A curly bracket is a closed side, the angle bracket is an open side. The following examples show how bounded and half-bounded, (half-)open and closed intervals can be specified.

```
1=:=5   {1,5}     means 1<=x<=5   bounded closed
1<:<5   <1,5>     means 1<x<5     bounded open
1=:<5   {1,5>     means 1<=x<5    bounded right-open
1<:=5   <1,5}     means 1<x<=5    bounded left-open
1=:  {1,}  {1,>   means 1<=x      left-bounded closed
1<:  <1,}  <1,>   means 1<x       left-bounded open
:=5  {,5}  <,5}   means x<=5      right-bounded closed
:<5  {,5>  <,5>   means x<5       right-bounded open
```

It is very important to note that the 2nd form results in discrete values, while the 3rd form results in a continuous interval.

Each element in a set can have its own form, i.e., one element can be a single value while another can be an interval. If a set consists of single or bounded discrete `start:end:incr` values only, the set will be expanded to an array. This makes it possible for array operators and functions (like `mean`) to be applied to such sets. E.g.

```
WHERE column > mean([10,30:100:5])
```

Another form of constructing a set is using a subquery as shown below.

4.9 Subqueries

As in SQL it is possible to create a set from a subquery. A subquery has the same syntax as a main query, but has to be enclosed in square brackets or parentheses. Basically it looks like:

```
SELECT FROM maintable WHERE time IN
    [SELECT time FROM othertable WHERE windspeed < 5]
```

The subquery on `othertable` results in a constant set containing the times for which the windspeed matches. Subsequently the main query is executed and selects all rows from the main table with times in that set. Note that like other bounded sets this set is transformed to a constant array, so it is possible to apply functions to it (e.g. min, mean).

```
SELECT FROM maintable WHERE time IN
    [SELECT time FROM othertable WHERE windspeed <
         mean([SELECT windspeed FROM othertable])]
```

This contains another subquery to get all windspeeds and to take the mean of them. So the first subquery selects all times where the windspeed is less than the average windspeed.
A subquery result should contain only one column, otherwise an exception is thrown.

It may happen that a subquery has to be executed twice because 2 columns from the other table are needed. E.g.

```
SELECT FROM maintable WHERE any(time >=
    [SELECT starttime FROM othertable WHERE windspeed < 5]
                              && time <=
    [SELECT endtime FROM othertable WHERE windspeed < 5])
```

In this case the other table contains the time range for each windspeed. For big tables it is expensive to execute the subquery twice. A better solution is to store the result of the subquery in a temporary table and reuse it.

```
SELECT FROM othertable WHERE windspeed < 5 GIVING tmptab
SELECT FROM maintable WHERE any(time >=
    [SELECT starttime FROM tmptab]
                              && time <=
    [SELECT endtime FROM tmptab])
```

However, this has the disadvantage that the table `tmptab` still exists after the query and has to be deleted explictly by the user. Below a better solution for this problem is shown.

TaQL has a few extensions to support tables better, in particular the AIPS++ measurement sets.

1. The temporary problem above can be circumvented by using the ability to use a `SELECT` expression in the `FROM` clause. E.g.

```
SELECT FROM maintable,
    [SELECT FROM othertable WHERE windspeed < 5] tmptab
    WHERE any(time >= [SELECT starttime FROM tmptab]
           && time <= [SELECT endtime FROM tmptab])
```

However, below a even nicer solution is given.

2. The time range problem above can be solved elegantly by using a set as the result of the subquery. Instead of a table name, it is possible to give an expression in the GIVING clause (as mentioned in the syntax section). E.g.

   ```
   select from MY.MS where TIME in
       [select FROM OTHERTABLE where WINDSPEED < 5
            giving [TIME-INTERVAL/2 =:= TIME+INTERVAL/2]]
   ```

   The set expression in the GIVING clause is filled with the results from the subquery and used in the main query. So if the subquery results in 5 rows, the resulting set contains 5 intervals. Thereafter the resulting intervals are sorted and combined where possible. In this way the minimum number of intervals have to be examined by the main query.

3. In AIPS++ the other table will often be the name of a subtable, which is stored in a table or column keyword of the main table. The standard keyword syntax can be used to indicate that the other table is the table in the given keyword. Note that for a table keyword the `::` part has to be given, otherwise the name is treated as an ordinary table name. E.g.

   ```
   select from MY.MS where TIME in
       [select TIME from ::WEATHER where WINDSPEED < 5]
   ```

   In this example the other table is a subtable of table `my.ms`. Its name is stored in keyword `WEATHER` of `my.ms`.

4. Often the result of a query on a subtable of a measurement set is used to select columns from the main table. However, several subtables do not have an explicit key, but use the row number as an implicit key. The function `ROWID()` can be used to return the row number as the subtable query result. E.g.

```
select from MY.MS where DATA_DESC_ID in
    [select from ::DATA_DESCRIPTION where
       SPECTRAL_WINDOW_ID in [0,2,4] giving [ROWID()]]
```

Note that the function ROWNUMBER cannot be used here, because it will give the row number in the selection and not (as ROWID does) the row number in the original table. Furthermore, ROWID gives a 0-relative row number which is needed to be able to use it as a selection criterium on the 0-relative values in the measurement set.

5 Some further remarks

5.1 Time/position considerations

A position constant (e.g. 3h4m) is converted to radians, so it can be used easily in functions like SIN. Since a time constant has exactly the same format, it is also converted to radians, while the user may expect it to be expressed in seconds. The user has to take this into account if times are used in a comparison. For example, timecol > 3h4m is only correct if timecol has unit radians.
To make life easier the function TIME results in a value in radians, so it can be used directly in a comparison. For example, TIME(datecolumn) > 3h4m.
In the future this may change if units are handled in their full glory.

5.2 Optimization

A lot of development work can be done to improve the query optimization. At this stage a few optimizations are done.

- Constant subexpressions are calculated only once. E.g. in COL*sin(180/pi) the part sin(180/pi) is evaluated once.

- If a subquery generates intervals of doubles or dates, overlapping intervals are combined and eliminated. E.g.

  ```
  select from GER.MS where TIME in [select from ::POINTING where
  sumsqr(DIRECTION[1])>0 giving [TIME-INTERVAL/2=:=TIME+INTERVAL/2]]
  ```

  can generate many identical or overlapping intervals. They are sorted and combined where possible to make the set as small as possible.

The user can optimize a query by specifying the expression carefully. When using operator || or &&, attention should be paid to the contents of the left and right branches. Both operators evaluate the right branch only if needed, so if possible the left branch should be the shortest one, i.e., the fastest to evaluate.

The user should also use functions, operators, and subqueries in a careful way.

- `SQUARE(COL)` is (much) faster than `COL^2` or `POW(COL,2)`, because SQUARE is faster. It is also faster than `COL*COL`, because it accesses column `COL` only once.
  Similarly `SQRT(COL)` is faster than `COL^0.5` or `POW(COL,0.5)`
- `SQUARE(U) + SQUARE(V) < 1000^2` is considerably faster than `SQRT(SQUARE(U) + SQUARE(V)) < 1000`, because the `SQRT` function does not need to be evaluated for each row.
- `TIME IN [0 <:< 4]` is faster than `TIME>0 && TIME<4`, because in the first way the column is accessed only once.
- Returning a column from a subquery can be done directly or as a set. E.g.

  ```
  SELECT FROM maintable WHERE time IN
      [SELECT time FROM othertable WHERE windspeed < 5]
  ```

  could also be expressed as

  ```
  SELECT FROM maintable WHERE time IN
      [SELECT FROM othertable WHERE windspeed < 5 GIVING [time]]
  ```

  The latter (as a set) is slower. So, if possible, the column should be returned directly. This is also easier to write.
  An even more important optimization for this query is writing it as:

  ```
  SELECT FROM maintable WHERE time IN
      [SELECT DISTINCT time FROM othertable WHERE windspeed < 5]
  ```

  Using the DISTINCT qualifier has the effect that duplicates are removed which often results in a much smaller set.
- Testing if a subquery contains at least N elements can be done in two ways:

```
  count([select column from table where expression]) >= N
and
  exists (select from table where expression limit N)
```

The second form is by far the best, because in that case the subquery will stop the matching process as soon as N matching rows are found. The first form will do the subquery for the entire table.
Furthermore in the first form a column has to be selected, which is not needed in the second form.

- Sometimes operator `IN` and function `ANY` can be used to test if an element in an array matches a value. E.g.

  ```
    WHERE any(arraycolumn == value)
  and
    WHERE value IN arraycolumn
  ```

  give the same result. Operator `IN` is faster because it stops when it finds a match. If using `ANY` all elements are compared first and thereafter `ANY` tests the resulting bool array.

- It was already shown in the indexing section that indexing arrays should be done with care.

6 Modifying a table

Usually TaQL will be used to get a subset from a table. However, as described in the first sections, it can also be used to change the contents of a table using the UPDATE, INSERT, or DELETE command. Note that a table has to be writable, otherwise those commands exit with an error message.

6.1 UPDATE

```
UPDATE table_list SET update_list [WHERE ...] [ORDERBY ...]
                                  [LIMIT ...] [OFFSET ...]
```

updates selected rows in a table.
`update_list` is a comma-separated list of `column=expression` parts. Each part tells to update the given column using the expression. Both scalar and array columns are supported. E.g.

```
UPDATE vla.ms SET ANTENNA1=ANTENNA1-1, ANTENNA2=ANTENNA2-1
```

to make the antenna numbers zero-based if accidently they were written one-based.
If an array gets an array value, the shape of the array can be changed (provided it is allowed for that table column). Arrays can also be updated with a scalar value causing all elements in the array to be set to that scalar value.

It is also possible to update part of an array using array indexing. E.g.

```
UPDATE vla.ms SET FLAG[1,1]=T
UPDATE vla.ms SET FLAG[1,]=T
```

The first example sets only the first array element, while the second one sets an entire row in the array. The second example also shows that it is possible to set an array to a scalar value.

Type promotion and demotion will be done as much as possible. For example, an integer column can get the value of a double expression (which will be truncated).

Note that if multiple `column=expression` parts are given, the columns are changed in the order as specified in the update-list. It means that if an updated column is used in an expression for a later column, the new value is used when evaluating the expression. E.g. in

```
UPDATE vla.ms SET DATA=DATA+1, SUMD=sum(DATA)
```

the `SUMD` update uses the new `DATA` values.

6.2 INSERT

The `INSERT` command adds rows to the table. It can take two forms:

```
INSERT INTO table_list [(column_list)] VALUES (expr_list)
INSERT INTO table_list [(column_list)] SELECT_command
```

The first form adds one row to the table and puts the values given in the expression list into the columns given in the column list. If the column list is not given, it defaults to all stored columns in the table in the order as they appear in the table description. Each expression in the expression list can be as complex as needed; for example, a subquery can also be given. Note that a subquery is evaluated before the new row is added, so the new row is not taken into account if the subquery is done on the table being modified. It should be clear that the number of columns has to match the number of

expressions.
Note that row cells not mentioned in the column list, are not written, thus may contain rubbish.
The data types of expressions and columns have to match in the same way as for the UPDATE command; values have to be promotable or demotable to the column data type.
For example:

```
INSERT INTO my.ms (ANTENNA1,ANTENNA2) VALUES (0,1)
```

adds one row, puts 0 in ANTENNA1 and 1 in ANTENNA2.

The second form evaluates the SELECT command and copies the rows found in the selection to the table being modified (which is given in the INTO part). The columns used in the modified table are defined in the column list. As above, they default to all stored columns. The columns used in the selection have to be defined in the SELECT part of the SELECT command. They also default to all stored columns.
For example:

```
INSERT INTO my.ms select FROM my.ms
```

appends all rows and columns of `my.ms` to itself. Please note that only the original number of rows is copied.

```
INSERT INTO my.ms (ANTENNA1,ANTENNA2) select ANTENNA2,ANTENNA1
 FROM other.ms WHERE ANTENNA1>0
```

copies rows from `other.ms` where ANTENNA1>0. It swaps the values of ANTENNA1 and ANTENNA2. All other columns are not written, thus may contain rubbish.

6.3 DELETE

```
DELETE FROM table_list
  [WHERE ...] [ORDERBY ...] [LIMIT ...] [OFFSET ...]
```

deletes some or all rows from a table.

```
DELETE FROM my.ms WHERE ANTENNA1>13 OR ANTENNA2>13
```

deletes the rows matching the WHERE expression.
If no selection is done, all rows will be deleted.
It is possible to specify more than one table in the FROM clause to be able to use, for example, keywords from other tables. Rows will be deleted from the first table mentioned in the FROM part.

7 Creating a new table

TaQL can be used to create a new table. The data managers to be used can be given in full detail.

The syntax is:

```
CREATE TABLE tablename colspec DMINFO datamans
```

The columns are defined in the `colspec` part. If not given, an empty table is created.
Data managers can be defined in the `datamans` part.

The `colspec` part can optionally be enclosed in square brackets or parentheses (for SQL compatibility). It is a comma separated list of column specifications. Each specification looks like:

```
columnname datatype [NDIM=n, SHAPE=[d1,d2,...],
                     DMTYPE='s',DMGROUP='s',COMMENT='s']
```

The possible data type strings are given in section Data Types. The part enclosed in square brackets is optional. Zero or more of these keywords can be used. It makes it possible to define array columns and/or default data manager to be used. The square brackets are optional if only one such keyword is used.
`NDIM=n` defines if the column contains scalars or arrays. A negative value means a scalar (unless shape is also given). A value 0 means an array of any dimensionality. A positive value means an array with the given dimensionality.
`SHAPE=[d1,d2,...]` makes it possible to define the exact array shape. If given and if NDIM is positive, they should be consistent.
`DMTYPE, DMGROUP, COMMENT` have string values, thus quotes have to be used. DMTYPE and DMGROUP are rather specific and are for the expert user.

The `dataman` part makes it possible for the expert user to define the data managers to be used by columns. It is a comma separated list of data manager specifications looking like the output of the `table.getdminfo` command in glish. Each specification has to be enclosed in square brackets. For example:

```
dminfo [NAME="ISM1",TYPE="IncrementalStMan",COLUMNS=["col1"]],
       [NAME="SSM1",TYPE="StandardStMan",
        SPEC=[BUCKETSIZE=1000],COLUMNS=["col2","col3"]]
```

The case of the keyword names used (e.g. NAME) is important. They have to be given in uppercase. The following keywords can be given:

`NAME` defines the unique name of the data manager.
`TYPE` defines the type of data manager.
`SPEC` is a list of keywords giving the characteristics of the data manager. This is highly data manager type specific.
`COLUMNS` is a list of column names defining all columns that have to be bound to the data manager.

8 Calculations on a table

TaQL can be used to get derived values from a table by means of an expression. The expression can result in any data type and value type. For example, if the expression uses an array column, the result might be a vector of arrays (an array for each row). If the expression uses a scalar column, the result might be a vector of scalars or even a single scalar if a reduction function like SUM is used.

The exact syntax is:

```
CALC [FROM table_list CALC] expression
```

The part in square brackets can be omitted if no column is (directly) used in the expression. The examples will make clear what that means.

```
CALC 1+2
```

is a simple expression not using a table. It shows how the CALC command can be used as a desk calculator.

```
CALC FROM mytable CALC mean(column1+column2)
```

gives a vector of scalars containing the mean per row.

```
CALC sum([SELECT FROM mytable GIVING [mean(column1+column2)]])
```

gives a single scalar giving the sum of the means in each row. Note that in this command the CALC command does not need the FROM clause, because it does not use a column itself. Columns are only used in the nested query which has a FROM clause itself.

9 Examples

9.1 Selection examples

9.1.1 Reference table results

`SELECT FROM mytable WHERE column1 > 0`
selects the rows in which the value of column1 > 0

`SELECT column0,column1 FROM mytable`
selects 2 columns from the table.

`SELECT column0,column1 FROM mytable WHERE column1>0`
is a combination of the previous selections.

```
SELECT FROM [SELECT FROM mytable ORDERBY column0 DESC]
   WHERE rownumber()<=10
```
selects the 10 highest values of `column0`.

`SELECT FROM mytable ORDERBY column0 DESC LIMIT 10`
is a more elegant solution using the newer LIMIT clause.

```
SELECT FROM mytable ORDERBY column0 DESC GIVING outtable
   SELECT FROM outtable WHERE rownumber()<=10
```
is the SQL-like solution for the previous problem. It is less elegant, because it requires two steps.

```
SELECT FROM mytable WHERE column0 IN
   [SELECT column0 FROM mytable ORDERBY column0 DESC][1:10]
```
is similar to above, but can select more values if there happen to be several equal values.

```
SELECT FROM some.MS WHERE
   near(TIME/(24*3600),MJD(1999/03/30/17:27:15))
```
selects the rows with the given time from a MeasurementSet.
Note that the TIME is stored in seconds; the division converts it to days.

```
SELECT FROM some.MS where TIME/(24*3600) in
   [{MJD(1999/03/30/17:27:15),MJD(1999/03/30/17:29:15)}]
```
selects the rows in the given closed time interval.

```
SELECT FROM some.MS where TIME/(24*3600) in
   [MJD(1999/03/30/17:27:15),MJD(1999/03/30/17:29:15)]
```

selects the rows having one of the given times.
Note the difference with the previous example where an interval was given. Here a set of two individual time values is given.

`SELECT FROM resource.table WHERE any(PValues == pattern('synth*'))`
selects the rows in which an element in array `PValues` matches the given regular expression.

`SELECT FROM table WHERE ntrue(flags) >= 3`
selects rows where at least 3 elements of array `flags` are set.

`SELECT FROM book.table WHERE nelements(author) > 1`
selects books with more than 1 author.

`SELECT FROM my.ms WHERE any(ANTENNA1==[0,0,1] && ANTENNA2==[1,3,2])`
selects the antenna pairs (baselines) 0-1, 0-3, and 1-2.
Note that the two comparisons result in a boolean vector. If a bool in the and-ed vectors is true, that baseline matches.

`SELECT FROM mytable WHERE cos(0d1m) < sin(52deg) * sin(DEC) + cos(52deg) * cos(DEC) * cos(3h30m - RA)`
selects observations with an equatorial position (in say J2000) inside a cone with a radius of 1 arcmin around (3h30m, 52deg). To find them the condition DISTANCE<=RADIUS must be fulfilled, which is equivalent to COS(RADIUS)<=COS(DISTANCE).

`SELECT FROM mytable WHERE [RA,DEC] INCONE [3h30m, 52deg, 0d1m]`
does the same as above in an easier (and faster) way.

`SELECT FROM mytable WHERE object == pattern("3C*") && [RA,DEC] INCONE [3h30m, 52deg, 0d1m]`
finds all 3C objects inside that cone.

`select from MY.MS where DATA_DESC_ID in [select from ::DATA_DESCRIPTION where SPECTRAL_WINDOW_ID in [0,2,4] giving [ROWID()]]`
finds all rows in a measurement set matching the given spectral windows.

```
select from MY.MS where TIME in
   [select from ::SOURCE where REST_FREQUENCY < 180000000.
   giving [TIME-INTERVAL/2 =:= TIME+INTERVAL/2]]
```

finds all rows in a measurement set observing sources with a rest frequency less than 180 Mhz.

```
select from MS,
   [select from MS where sumsqr(UVW[1:2]) < 625] as TIMESEL
   where TIME in [select distinct TIME from TIMESEL]
   && any([ANTENNA1,ANTENNA2] in [select from TIMESEL giving
   [iif(UVW[3] < 0, ANTENNA1, ANTENNA2)]])
```

finds all antennas which are shadowed at a given time.
The query in the FROM command finds all rows where an antenna is shadowed (i.e. its UV-distance less than 25 meters) and creates a temporary table. This selection is done in the FROM command, otherwise two 2 equal selections are needed in the main WHERE command.

```
select from MS
   where DATA_DESC_ID in [select from ::DATA_DESCRIPTION
   where SPECTRAL_WINDOW_ID in [select from ::SPECTRAL_WINDOW
   where NET_SIDEBAND==1 giving [ROWID()]] giving [ROWID()]]
```

finds all rows in the MeasurementSet with the given NET_SIDEBAND.
The MeasurementSet uses a table to map spectral-window-id to data-desc-id. Hence two nested subqueries are needed.

9.1.2 Plain table results

```
SELECT column0+column1 FROM mytable
```

creates a table of 1 column with name Col_1. Its data type is on the expression data type.

```
SELECT column0+column1 Res I4 FROM mytable
```

creates a table of 1 column with name `Res`. Its data type is 4 byte signed integer.

```
SELECT colx colx R4 FROM mytable
```

creates a table of 1 column with name `colx`. The sole purpose of this selection is to convert the data type of the column.

```
SELECT means(DATA,2) AS DATA_MEAN C4 FROM my.ms
```

creates a table of 1 column with name `DATA_MEAN`. Column DATA in a AIPS++ MeasurementSet is a 2-dimensional array with axes polarization and frequency. This command calculates and stores the mean in each polarization. If no data type was given, the means would have been stored as double precision complex (which is the expression data type).

9.2 Modification examples

`update MY.MS set VIDEO_POINT=MEANS(DATA,2) where isdefined(DATA)`

sets the VIDEO_POINT of each correlation to the mean of the DATA for that correlation. Note that the 2 indicates averaging over the second axis, thus the frequency axis.

`update MY.MS set FLAG_ROW=T where isdefined(FLAG) && all(FLAG)`
sets FLAG_ROW in the rows where the entire FLAG array is set.

`delete from MY.MS where FLAG_ROW`
deletes all flagged rows.

`insert into MY.MS select from OTHER.MS where !FLAG_ROW`
copies all unflagged rows from OTHER.MS to MY.MS.

```
insert into MY.MS/DATA_DESCRIPTION
   (SPECTRAL_WINDOW_ID,POLARIZATION_ID,FLAG_ROW)
   values (1,0,F)
```
adds a row to the DATA_DESCRIPTION subtable and initializes it.

9.3 Table creation examples

`create table mytab (col1 I4, col2 I4, col3 R8)`
creates table `mytab` of 3 scalar columns.

`create table mytab`
creates an empty table.

`create table mytab colarr R4 ndim=0`
creates a table of 1 array column with arbitrary dimensionality.

`create table mytab colarr R4 [shape=[4,128],  dmtype='TiledColumnStMan']`

creates a table of 1 array column with the given shape. The column is stored with the TiledColumnStMan storage manager using its default settings.

```
create table mytab colarr R4 shape=[4,128]  dminfo [TYPE='TiledColumnStMan',
  NAME='TCSM',  SPEC=[DEFAULTTILESHAPE=[4,32,64]],  COLUMNS=['colarr']]
```

creates a table of 1 array column with the given shape. The column is stored with the TiledColumnStMan storage manager using the given settings.

9.4 Calculation examples

```
calc 1+2
```

uses TaQL as a desktop calculator.

```
calc 7-Apr-2007 - 20-Nov-1979
```

calculates the number of days between these dates.

```
calc (1-1-2006 - 1-1-1950)%365
```

calculates the number of leap days in this time span.

```
calc sum([select from MY.MS giving [ntrue(FLAG)]])
```

determines the total number of flags set in the measurement set.

```
calc from [select from MY.MS where ANTENNA1==0]
  calc mean(abs(DATA))
```

calculates for each row the mean of the data for the selected subset of the measurement set.

```
calc mean([select from MY.MS where ANTENNA1==0
  giving [mean(abs(DATA))]])
```

looks like the previous example. It, however, calculates the mean of the mean of the data in each row for the selected subset of the measurement set.

```
calc max([select from MY.MS where isdefined(DATA)
  giving [max(abs(VIDEO_POINT-MEANS(DATA,2)))]])
```

shows the maximum absolute difference between VIDEO_POINT of each correlation and the mean of the DATA for that correlation. Note that the 2 indicates averaging over the second axis, thus the frequency axis.

10 Interface to TaQL

There exists a user and a programmer interface to TaQL. Glish functions form the user interface, while C++ functions and classes form the programmer interface.

- The main TaQL interface in Glish is formed by the `query` function in module `table` (in script `table.g`). The function can be used to compose and execute a TaQL command using the various (optional) arguments given to the `query` function. E.g.

  ```
  tab := table('mytable')
  seltab1 := tab.query ('column1 > 0')
  seltab2 := seltab1.query (query='column2>5',
                              sortlist='time',
                              columns='column1,column2',
                              name='result.tab')
  ```

  The first command opens the table `mytable`. The second command does a simple query resulting in a temporary table. That temporary table is used in the next command resulting in a persistent table. The latter function call is transformed to the TaQL command:
  `SELECT column1,column2 FROM $1 WHERE column2>5`
  `ORDERBY time GIVING result.tab`
  During execution $1 is replaced by table `seltab1`.
  Note that the `name` argument generates the `GIVING` part to make the result persistent.

  It is possible to embed glish variables and expressions in a TaQL command using the syntax `$variable` and `$(expression)`. A variable can be a standard numeric or string scalar or vector. It can also be a table tool. An expression has to result in a numeric or string scalar or vector. E.g

  ```
  tab := table('mytable')
  coldata := tab.getcol ('col');
  colmean := sum(coldata) / len(coldata);
  seltab1 := tab.query ('col > $colmean')
  seltab2 := tab.query ('col > $(sum(coldata)/len(coldata))')
  seltab3 := tab.query ('col > mean([SELECT col from $tab])')
  ```

These three queries give the same result.
The substitution mechanism is described in more detail in the `substitute` functions of module `utility` of the User Reference Manual.
The substitution mechanism uses the eval function in glish. As of 28-Feb-2001 eval only looks at global variables. This means that in a function one needs to create a global variable (with a unique name) if the variable is to be used in a TaQL command. The global variable should be deleted at the end of the function. The name can be made unique by using the function name as a suffix. E.g.:

```
myfunc := function() {
  tab := table('mytable')
  global coldata_in_myfunc;
  coldata_in_myfunc := tab.getcol ('col');
  seltab := tab.query ('col > $(sum(coldata)/len(coldata))')
  symbol_delete ('coldata_in_myfunc');
}
```

The other function that can be used is `tablecommand`. The full TaQL command has to be given to that command. The result is a table tool. E.g.

```
t := tablecommand('select from GER.MS where ANTENNA1==1');
```

- The C++ interface consists of 2 parts.

  1. The function `tableCommand` in TableParse.h can be used to execute a TaQL command. The result is a Table object. E.g.

     ```
     Table seltab1 = tableCommand
             ("select from mytable where column1>0");
     Table seltab2 = tableCommand
             ("select column1,column2 from $1 where column2>5"
              " orderby time giving result.tab", seltab1);
     ```

     These examples do the same as the Glish ones shown above.
     Note that in the second function call the table name `$1` is replaced by the object `seltab1` passed to the function.

2. The other interface is a true C++ interface having the advantage that C++ variables can be used. Class Table contains functions to sort a table or to select columns or rows. When selecting rows class TableExprNode (in ExprNode.h) has to be used to build a WHERE expression which can be executed by the overloaded function operator in class `Table`. E.g.

```
Int limit = 0;
Table tab ("mytable");
Table seltab = tab(tab.col("column1") > limit);
```

   does the same as the first example shown above. See classes Table, TableExprNode, and TableExprNodeSet for more information on how to construct a WHERE expression.

11 Possible future developments

In the near or far future TaQL will be enhanced by adding new features and by doing optimizations.

- If a branch in an `OR` or `AND` node is a constant scalar, it might be possible to reduce the node to one of its branches.
- Optimize by removing an `IN` node if the righthand operand is empty. This cannot be done if the lefthand operand is variable shaped.
- Handle invalid subexpressions (e.g. exceeding array bounds) as undefined values which can be tested with the function ISDEFINED.
- Interpret column units defined in a column keyword UNIT.
- Maybe functions to convert positions from one equinox or system to another (using the Measures system).
- Optimize using a B+ tree index.
- Expressions in the selected column-list to calculate new columns. E.g. `SELECT SQUARE(UVW[1])+SQUARE(UVW[2]) UVWNORM FROM ...` where `UVWNORM` is the name for that new column.