NOTE 199 – Table Query Language

Ger van Diepen, ASTRON Dwingeloo

2013 Aug 19

Abstract

The Table Query Language (TaQL) is an SQL-like high level language to do operations like selection, sort, and update on a casacore table. It is a very versatile language with full support for table columns containing array data. It has inherent support for masked arrays, units, and astronomical coordinates. 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++, Python, and Glish is available.

Contents

1 Introduction

The Table Query Language (TaQL, rhymes with bagel (though some 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 units, extended regular expressions, and many functions. User defined functions written in C++ are also supported. 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. Some familiarity with SQL makes it easier to understand the TaQL syntax. 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 it is also possible to make a deep copy and create 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.
- A very rich set of mathematical and other functions.
- Any operand can be a scalar or an N-dimensional array. Many reduction functions can be applied to arrays.
- Arrays can optionally be masked.
- Full support of units and automatic conversion of units.
- Support of various types of patterns/regular expressions and maximum string distances (Levensthein (aka Edit) distance).
- 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 a linear table search is done. Because data are stored column-wise, a linear search is usually very fast, even for very large tables.
- Limited support for joins (only implicit joins on row number).
- In the SELECT command GROUPBY and HAVING are not supported yet.
- The COUNT command exists to count the occurrences of column values.
- 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.
- TaQL can be used from languages with different conventions, for example the order of array axes. Therefore it is possible to set the language style to be used.

- The language can be extended by means of User Defined Functions. Some standard UDFs exist to deal with measure conversions (for directions, epochs, positions, and stokes).

TaQL has a keyword that makes it possible to time the various parts of a TaQL command.

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

2 TaQL Commands

2.1 Command summary

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 [AS type] | set]
```

  It can be used to get an optionally sorted subset from a table. It can also be used to do a subquery (see section 4.11 for more information on subqueries).

- 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.

- counting

  ```
  COUNT [column_list] FROM table_list [WHERE ...]
  ```

  It can be used to count occurrences of column values. It is similar to the GROUPBY clause in SQL which is not impemented yet.

- calculation

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

  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. Optionally specific data manager info can be given.

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.
Section 6 (Modifying a table) explains the UPDATE, INSERT, and DELETE commands in more detail. The CREATE TABLE command is explained in section 7 (Creating a table).
Section 8 (Counting in a table) explains the COUNT command in more detail.
Section 9 (Calculations on a table) explains the CALC command in more detail.

2.2 Using a style

TaQL can be used from different languages, in particular Python and Glish. Each has its own conventions breaking down into three important categories:

- 0-based or 1-based indexing.
- Fortran-order or C-order of arrays.
- Inclusive or exclusive end in `start:end` ranges.

The user can set the style (convention) to be used by preceeding a TaQL statement with

```
USING STYLE value, value, ...
```

The possible (case-independent) values are:

- `BASE0` or `BASE1` telling the indexing style.
- `ENDEXCL` or `ENDINCL` telling the range style.
- `CORDER` or `FORTRANORDER` telling the array style.

- `PYTHON` which is equivalent to `BASE0,ENDEXCL,CORDER`
- `GLISH` which is equivalent to `BASE1,ENDINCL,FORTRANORDER`

If multiple values are given for a category, the last one will be used. The default style used is `GLISH`, which is the way TaQL always worked before this feature was introduced.

It is important to note that the interpretation of the axes numbers depends on the style being used. E.g. when using glish style, axes numbers are 1-based and in Fortran order, thus axis 1 is the most rapidly varying axis. When using python style, axis 0 is the most slowly varying axis. Casacore arrays are in Fortran order, but TaQL maps it to the style being used. Thus when using python style, the axes will be reversed (data will not be transposed).

The style feature has to be used with care. A given TaQL statement will behave differently when used with another style.

2.3 Timing

It is possible to time a TaQL command by giving the case-insensitive keyword `TIME` after the optional style keywords described above. If given, the total execution time and the times needed for various parts of the TaQL command will be shown on stdout. For example:

```
time select distinct ANTENNA1,ANTENNA2
from ~/3C343.MS where any(FLAG)'

  Where          2.87 real    2.16 user    0.69 system
  Projection        0 real       0 user       0 system
  Distinct       0.18 real    0.16 user    0.03 system
 Total time      3.07 real    2.33 user    0.72 system
```

shows the time to do the where part (i.e. row selection on FLAG), projection (selection of columns), and distinct (unique column values).

2.4 Reserved words

TaQL uses the following words as part of its language.

```
  ALL           AND      AS        ASC
  BETWEEN
  CALC          CREATETABLE
  DELETE        DESC     DISTINCT  DMINFO
  EXCEPT        EXISTS
  F             FALSE    FROM
  GIVING        GROUPBY
  HAVING
  IN            INCONE   INSERT    INTERSECT  INTO
  JOIN
  LIKE          LIMIT
  MINUS
  NODUPLICATES  NOT
```

```
OFFSET          ON        OR          ORDERBY
SAVETO          SELECT    SET
T               TRUE
UNION           UNIQUE    UPDATE      USINGSTYLE
VALUES
WHERE
XOR
```

These words are reserved. Note that the words in the TaQL vocabulary are case insensitive, thus the lowercase (or any mixed case) versions are also reserved.

The reserved words cannot directly be used as column name, keyword name, or unit. However, a reserved word can be used by escaping it with a backslash like `\AS`. When reading further, the meaning of

```
   \IN  \in  IN [3mm,4mm]
column unit IN    set
```

might come clear. It means: use unit `in` (inch) for column `IN` and test if it is in the given set.
Note this is unlike SQL where quotes have to be used to use a reserved word as a column name.

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.

There is no explicit JOIN clause, but it is possible to equi-join tables on row number. Such tables have to have the same number of rows.
One can also join , for example, the main table of a MeasurementSet with a subtable like the ANTENNA table using a subquery.
Joins are explained further in section 5.

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 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
```

Apart from giving exact column names, it is also possible to use wildcards by means of a UNIX filename-like pattern (like `p/pattern/`) or a regular expression (like `f/regex/` for a full match or `m/regex/` for a partial match). They can be suffixed with an `i` indicating case-insensitive matching. See section 4.3.6 for a discussion of these constants. The operator `~` needs to be given before the pattern or regex to indicate that columns have to be included. Thereafter operator `!~` can be used with another pattern or regex to remove columns. Such an excluding pattern or regex only removes columns from the wildcarded columns before it until the latest non-wildcarded column.
A special pattern is * (which is the same as `p/*/`). For example:

```
select *, !~p/*_DATA/ from 3C343.MS
```

selects all columns except the ones ending in `_DATA`.

```
select ~m/DATA/, !~p/*_DATA/ from 3C343.MS
```

selects columns with a name containing `DATA` except the ones ending in `_DATA`.

```
select CORRECTED_DATA, *, !~p/*_DATA/ from 3C343.MS
or
select *, !~p/*_DATA/, CORRECTED_DATA from 3C343.MS
```

does select the `CORRECTED_DATA` column.
Note it is not possible to change the name or data type of wildcarded columns.

It is also 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 plain 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 unit conversion can be (part of) an expression. For example:

```
select TIME d AS TIMEH from my.ms
```

to store the time in unit `d` (days). Units are discussed in section 4.9.

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 starting with a parenthesis 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 table argument gives the name of the resulting table. It can be omitted if a memory table is created.

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

MEMORY to store the result in a memory table.

SCRATCH to store the result in a scratch table, possibly on disk.

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.

If the INTO (or GIVING) clause is not given, the query result will be written into a memory table. In this way queries done in a readonly directory will not fail because it cannot create a result table. However, if the result is expected to not fit in memory (which will seldomly be the case), type SCRATCH should be used to make it fit.

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. If the tablename contains a special character, the character can be escaped with a backslash or the

table name can be enclosed in single or double quotes.
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 very well possible to specify more tables in the table_list to execute a join or to get a keyword value from another table. 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.

Multiple table lists can also be used to join the tables on row number. E.g.

```
SELECT FROM mytable t1,othertable t2
  WHERE not all(near(t1.data, t2.data))
```

Such a command could be used to check if the data in mytable is about equal to the data in othertable. It checks if both tables have the same number of rows.

The full table_list syntax is:
`table_name1 [shorthand1], table_name2 [shorthand2], etc.`
The shorthand defaults to the table_name. A shorthand (alias) 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 four 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. Opening a subtable using a path name like `my.ms/ANTENNA` will fail if `my.ms` is a reference table instead of the original table. Therefore the path of a subtable can be given using colons instead of slashes like `my.ms::ANTENNA` which is a slight extension of specifying table names in the previous bullet.
   In this way a subtable can always be found, no matter where it is located.

3. 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. Besides the SELECT command the COUNT command can also be used. 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.

4. 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 Python, 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 section 4.

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/OFFSET expression

It indicates which of the matching and sorted rows should be selected. If not given, all of them are selected.
`LIMIT` and `OFFSET` are applied after `ORDERBY` and `SELECT DISTINCT`, so they are particularly useful in combination with those clauses to select, for example, the highest 10 values.

It can be given in two ways:

- In the semi-standard SQL way using `LIMIT N` to select N rows and/or `OFFSET M` to skip the first M rows. Similar to Python, N and M can be negative meaning they are counted from the end. E.g., `LIMIT -1` means all rows but the last.

- As a Python-style range using `LIMIT start:end:incr`, where the end is exclusive. Start defaults to 0, end to the number of rows, and incr to 1. As above, start and end can be negative to count from the end. The increment must be positive.

For example:

```
SELECT FROM my.tab ORDERBY DISTINCT TIME LIMIT 2 OFFSET 10
SELECT FROM my.tab ORDERBY DISTINCT TIME LIMIT 10:12
```

sorts uniquely by time, skips the first 10 rows, and selects the next two rows.

```
SELECT FROM my.tab LIMIT ::100
```

selects every 100-th row.

3.7 GIVING [table] [AS type] | set

It indicates that the ultimate result of the SELECT command should be written to a table (with the given name).
Another (more SQL compliant) way to define the output table is the INTO clause. See INTO for a more detailed description including the possible types.

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 expression is the basic building block of TaQL. They similar to expressions in other languages. An expression is formed by applying an operator or a function to operands which can be a table column or keyword, a constant, or a subexpression. An operand can be a scalar value or an array or set of various data types. The next subsections discuss them in detail.

An expression can be used in several places:

- In the WHERE clause where the result must be a boolan scalar value. It tells if a table row will be selected.
- As a sort key in the ORDERBY clause where the result must be a scalar value (numeric, bool, or string)
- As an element in the set in the GIVING clause. It must be a scalar value of any type except regex.
- As a scalar or array value in the INSERT and UPDATE command.

- As a column expression in column-list part of the SELECT command. The result can be a scalar or array value.
- As a scalar or array value in the CALC command.

The expression in the clause can be as complex as one likes using arithmetic, comparison, and logical operators, functions, and units. Parentheses can be used to group subexpressions.
The operands in an expression can be table columns, table keywords, constants, units, 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 logical values (true or T or false or F)

Integer integer numbers up to 64 bits

Double 64 bit floating point numbers including times/positions

Complex 128 bit complex numbers

String string values on which operator + can be used (concatenation).

Regex regular expressions can be used for string matching (see section 4.2). Maximum string distances can also be used in a way similar to regular expressions.

DateTime representing a date/time. There are several functions acting on a date/time. Operator + and - can be used on them.

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 will do the following automatic conversions:
- from Integer to Double or Complex.
- from Integer or Double to Complex.
- from String or Double to DateTime.

In this document some special data types are used when describing the functions.
- **Real** means Integer or Double.
- **Numeric** means Integer, Double, or Complex.
- **DNumeric** means Double or Complex.

TaQL supports any possible data type of a table column or keyword. In some commands (column list and CREATE TABLE) columns are created where it is possible to specify the data type of a column. The following case-insensitive values can be used to specify a type:

```
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 Regular Expressions and String Distances

TaQL supports the use of extended regular expressions and string distances. They can be specified in various ways as discussed in section 4.3.6. There are three basic types of regular expressions.

- An SQL-style pattern is quite simple. It has 2 special characters. The underscore (_) means a single arbitrary character and the percent (%) means zero or more arbitrary characters. Special characters can be escaped with a backslash to retain their normal meaning. For example:

  ```
  3c\_%
  ```

  matches `3c_` and `3c_xx`, but not `3caxx`.

- A UNIX-style pattern, as often used for wildcarded file names, is more powerful than the SQL-style pattern. It has a few special characters that can be escaped with a backslash.
  - The question mark (?) means a single arbitrary character.
  - The asterisk (*) means zero or more arbitrary characters. For example: `3c_*` does the same as the SQL-style pattern above.
  - Square brackets indicate a bracket expression (character choice). For example: `[ab]` matches `a` and `b`, but not `c`. A few special characters can be used in a bracket expression:
    * A leading `^` or `!` means negation. Thus `[!ab]` matches every character except a and b.
    * A minus sign indicates a range. For example `[0-9]` matches a digit or `[a-z]` matches a lowercase letter. If a minus sign cannot be interpreted as a range, it is a literal minus sign like in `[-ab]` or the second minus sign in `[a-z-A]`.
    * Posix character classes `[:xx:]` where xx can be:
      - **alpha** matching any letter
      - **lower** matching any lowercase letter
      - **upper** matching any uppercase letter
      - **alnum** matching any digit or letter
      - **digit** matching any digit
      - **xdigit** matching any hexadecimal digit (0-9a-fA-F)

- **space** matching any whitespace character
- **print** matching any printable character (alnum, punct, space)
- **punct** matching any non-alnum visible character (.,!? etc.)
- **graph** matching any visible printable character (alnum, punct)
- **cntrl** matching any control character.

For example `[_[:isalpha:]][_[:isalnum:]]*` to match variable names.

* A bracket expression cannot be empty, thus if] is the first character in the bracket expression, it is interpreted literally. Note that is also true if it is the first character after the negation character.
* A backslash in a character class is always interpreted literally, thus special characters cannot be escaped. However, as shown above they can always be placed such that they are interpreted literally.

– Braces can be used for a choice between (possible empty) multi-character strings separated by commas. Escape a comma or brace with a backslash to treat it literally. For example:
`*.{h,hpp,c,cc,cpp}`
It is fully nestable, thus choice strings can be patterns. For example:
`*.{[hc]{,pp},c}`
does the same as the example above. Note that the inner choice is between an empty string and `pp`.

- An awk/egrep-like extended regular expression is most powerful. A full explanation can be found on Wikipedia. Here only a summary of its special characters is given. They can be escaped using backslashes.

  – `.` matches any character.
  – `^` matches beginning of string.
  – $ matches end of string.
  – Square brackets for a bracket expression. It is the same as described above with the exception that ! cannot be used as negation character).
  – * matches zero or more occurrences of previous character or subexpression.
  – + matches one or more occurrences.
  – ? matches zero or one occurrence.
  – { and } for an interval giving minimum and maximum number of occurrences. For example:
  `[a-z]{3,5}` matches lowercase string with a minimum of 3 and maximum of 5 characters.
  `[a-z]{3}` matches exactly 3 characters.
  `[a-z]{3,}` matches at least 3 characters.
  `[a-z]{,5}` matches at most 5 characters.
  – `|` matches left or right substring
  – (and) to form subexpressions for operators like *.
  – `\1` till `\9` mean backreference to a subexpression (first one is `\1`). A string part matches if it is equal to the string part matching that subexpression. E.g. `(a*)x\1` matches x, axa, aaxaa, etc., but not axaa nor aaxa.

For example:

```
.*\.(h|hpp|c|cc|cpp)
.*\.[hc](pp)?|cc
```

do the same as the pattern examples above.

Furthermore it is possible to specify maximum string distances (known as Levensthein or Edit distance). It is explained in section 4.3.6.

```
column ~ d/string/ibnn
```

4.3 Constants

Scalar constants of the various data types can be formed in a way similar to Python and Glish. Array constants can be formed from scalar constants.

4.3.1 Bool

A Bool constant is the value `T` or `F` (both in uppercase) or the value `true` or `false` (any case).

4.3.2 Integer

An integer constant is a numeric value without decimal point or exponent. It can also be given as a hexadecimal value like `0xffff`.

4.3.3 Double (and time/position)

A floating-point constant is given with a decimal point and/or exponent. Both E and D as well as their lowercase versions can be used to specify an exponent. An integer number followed by a unit is also regarded as a double constant.
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 simple 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 HMS format. Minutes and/or seconds can be left out. E.g. `12h34m34.5` or `8h`
- A position in DMS format. Only seconds can be left out because `12d3` is the floating point constant 12000. E.g. `12d34m34.5` or `8d0m`
- A position as DMS 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`

Note that DMS positions and units have an ambiguity. `2d3m` will be seen as a position and not as 2000 meter (note that 2d3 is a floating point constant). Use `2.d3m` or `2d3 m` to indicate the meters case.

4.3.4 Complex

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

```
1.5 + 2j
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)
```

4.3.5 String

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:   ab'cdef"gh
```

4.3.6 Regular expression and String distance

A regular expression constant can be given directly or using a function.

- An SQL-style pattern can be given directly as a string constant preceeded by operator LIKE or NOT LIKE.
- A pattern or regular expression can be given like `x/expr/q` preceeded by operator `~` or `!~`. 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 expression (not even escaped with a backslash).
  The x denotes the type:
  - **p** means a pattern matching the full string.
  - **f** means a regular expression matching the full string.
  - **m** means a regular expression matching part of the string (a la Perl).

  The q denotes optional qualifiers. Currently only **i** is supported meaning a case-insensitive match. For example:

  ```
  name~p/3[cC]*/
  name ~ p%3c*%i
  lower(name) ~ p%3c*%
  name ~ m/^3c/i
  name ~ f/3c.*/i
  filename !~ p#/usr/*.{h,cc}#
  ```

All examples but the last one do the same: matching a name starting with 3c or 3C.
The last example shows a glob-style pattern to find files on `/usr` not ending in `.h` or `.cc`.

- Apart from these Perl-like specifications, a regular expression can also be formed by applying a function to a string constant. The operator = or != has to be applied to it.
  - Function `sqlpattern` treats its argument as an SQL-style pattern. For example:

    ```
    lower(name) LIKE '3c%'
    lower(name) = sqlpattern('3c%')
    ```

    do the same.
  - Function `pattern` treats its argument as a UNIX-style pattern.
  - Function `regex` treats its argument as a full regular expression.

  Case-insensitive matching can only be done as shown in the example above by downcasing the string to be matched.
  Please note that these functions are not limited to constants. They can also be used to form regular expressions from variables.

A maximum string distance constant can be specified in a similar way. Such a distance is known as the Levensthein or Edit distance. It is a measure of the similarity of strings by counting the minimum number of edits (deletions, insertions, substitutions, and swaps of adjacent characters) that need to be done to make the strings equal.

```
column ~ d/string/ibnn
```

This tests if the strings in the given column are within the maximum distance of the string given in the constant. The following qualifiers can be given (in any order):

- **i** means a case insensitive test.
- **b** means that blanks in the strings are ignored.
- **nn** is an integer value giving the maximum distance. If not given it defaults to `1 + len(string) / 3`.

4.3.7 Date/time

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 space, - 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 space, -, or / delimiter) is optional. The possible date formats are:
- YYYY/MM/DD or DD-MM-YYYY
- DD-MMMMMMMM-YY where the - is optional and MMMMMMM 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.

4.3.8 Arrays

N-dimensional arrays of all data types can be created with the exception of regular expressions.
It is possible to form a 1-dimensional array from a constant bounded discrete set. When needed such a set is automatically transformed to an array. E.g.

```
[1:10]
['str1', 'str2', 'str3']
'str' + ['1', '2', '3']
```

The first example results in an integer array of 10 elements with values 1..10. The others result in a string array of 3 elements. The second version already shows that strings can be concatenated (as explained further on).

Furthermore it is possible to use the `array` function to create an array of any shape. The values are given in the first argument as a scalar, srt, or another array. The shape is given in the latter

arguments as scalars or as a set. The array is initialized to the values given which are wrapped if the array has more elements.

```
array([1:10],10,4)
array([1:10], [10,4])
array(F,shape(DATA))
```

The first examples create an array with shape [10,4] containing the values 1..10 in each line. The latter results in a boolean array having the same shape as the DATA array and filled with False.

4.3.9 Masked Arrays

An array can have an optional mask. Similar to numpy's masked array, a mask value True means that the value is masked off, thus not taken into account in reduction functions like calculating the mean.
Note that this definition is the same as the FLAG column in a MeasurementSet, but is different from a mask in a casacore Image where True means good and False means bad.

All operations on arrays will take the possible mask into account. Reduction functions like `median` only use the unmasked array elements. Furthermore, partial reduction functions like `medians` will set an output mask element to True if the corresponding input array part has no unmasked elements.
Operators like + and functions like `cos` operate on all array elements. The mask in the resulting array is the logical OR of the input masks. Of course, the result has no mask if no input array has a mask.

A masked array is created by applying a boolean array to an array using the square brackets operator. For example:

```
DATA[FLAG]
DATA[DATA > 3*median(DATA)]
```

The first example applies the FLAG column in a MeasurementSet to the DATA column. The second example masks off high DATA values.

The function `arraydata`, `arraymask`, and `flatten` can be used to get the array data or mask. The last one flattens the array to a vector while removing all masked elements.

4.4 Table Columns

A table column can be used in a query by giving its name in the expression. 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.

A column can contain a scalar or an array value of any data type supported by the table system. It will be mapped to the available TaQL data types.

If the column keywords define a unit for the column, the unit will be used by TaQL.

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`.
A reserved word cannot be used directly as column name. It can, however, be used as a column name by escaping it with a backslash. E.g. `\IN`.
Note that in programming languages like C++ and Python a backslash itself has to be escaped by another backslash. E.g. in Python: `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.

4.5 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 the same 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.

4.6 Operators

TaQL has a fair amount of operators which have the same meaning as their C and Python counterparts. The operator precedence order is:

```
**
!  ~  +  -       (unary operators)
*  /  // %
+  -
&
^
|
```

```
== != >  >= <  <=  ~= !~= IN INCONE BETWEEN EXISTS LIKE  ~  !~
&&
||
```

Operator names are case-insensitive. For SQL compliancy some operators have a synonym.

```
==      =
!=      <>
&&      AND
||      OR
!       NOT
^       XOR
```

All operators can be used for scalars and arrays and a mix of them. Note that arrays of regular expressions cannot be used.

The following table shows all available operators and the data types that can be used with them.

Operator	Data Type	Description
`**`	numeric	power. It is right associative, thus `2**1**2` results in `2`.
`*`	numeric	multiplication
`/`	numeric	non-truncated division, thus `1/2` results in `0.5`
`//`	real	truncated division (a la Python) resulting in an integer, thus `1./2.` results in `0`
`%`	real	modulo; `3.5%1.2` results in `1.1`; `-5%3` results in `-2`
`+`	no bool	addition. If a date is used, only a real (converted to unit day) can be added to it. String addition means concatenation.
`-`	numeric,date	subtraction. Substracting a date from a date results in a real (with unit day). Subtracting a real (converted to unit day) from a date results in a date.
`&`	integer	bitwise and
`|`	integer	bitwise or
`^`, XOR	integer	bitwise xor
==, =	all	comparison for equal. The norm is used when comparing complex numbers.
>	no bool	comparison for greater
>=	no bool	comparison for greater or equal
<	no bool	comparison for less
<=	no bool	comparison for less or equal
!=, <>	all	comparison for not equal
`~=`	numeric	shorthand for the NEAR function with a tolerance of 1e-5
`!~=`	numeric	shorthand for NOT NEAR with a tolerance of 1e-5
`&&`, AND	bool	logical and
`||`, OR	bool	logical or
`!`, NOT	bool	logical not
`~`	integer	bitwise negation
`+`	numeric	unary plus
`-`	numeric	unary minus
`~`	string	test if string matches a regular expression constant.
`!~`	string	test if string does not match a regular expression constant.
LIKE	string	test if a string matches an SQL pattern.
NOT LIKE	string	test if string does not match SQL pattern.
IN	all	test if a value is present in a set of values, ranges, and/or intervals. (See the discussion of sets).
NOT IN	all	negation of IN
BETWEEN	no bool	`a BETWEEN b AND c` is similar to `a>=b AND a<=c` and `a IN [b=:=c]`
NOT BETWEEN	no bool	`a NOT BETWEEN b AND c` is negation of above.
INCONE		cone search. (See the discussion of cone search functions).
NOT INCONE		negation of INCONE
EXISTS		test if a subquery finds at least N matching rows. The value for N is taken from its 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 true if at least 2 matching rows in the ANTENNA table were found.
NOT EXISTS		negation of EXISTS

4.7 Sets and intervals

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

```
SELECT FROM table WHERE column IN [1,2,3]
```

The result of operator `IN` is true if the column value matches one of the values in the set. A set can contain any data type except a regex.

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 or convertible to a common data type. 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 array is also a set, so `IN` can also be used on an array like:

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

where `expr1` is the array result of some expression. 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 `==`.

The lefthand operand of the `IN` operator can also be an array or set. In that case its reult is a boolean array telling for each element in the lefthand operand if it is found in the righthand operand.

An element in a set can be more complicated than a single value. It 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 real or a datetime. Incr has to be a real. Some examples:

   ```
   1:10     means 1,2,...,9,10  (10 only when using glish style)
   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 data type real, 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 of set specification 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 described in section 4.11.

4.8 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,...]`. This syntax is similar to that used in Python or Glish. Taking a single element can be done as:

```
array[1, 2]
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]
                    (NB. shape is [2,5,7] for python style)
```

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 real value which will be truncated to an integer length.
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.

4.9 Units

Units can be given at many places in an expression; in fact, each subexpression can be ended with a unit meaning that the subexpression result gets that unit or will be converted to that unit. A simple unit (only letters) can always be given literally. A non-simple unit can be given literally if only containing digits, underscores and/or dots (e.g. `m2`, `fl_oz`. or `m.m`). Otherwise the unit has to be enclosed in single or double quotes (e.g. `'m/s'`) or the backslash has to be used as escape character (e.g. `\in` or `m\/s`).
Arguments to functions like `sin` will be converted to the appropriate unit (radians) as needed. In a similar way, the units of operands to operators like addition, will be converted as needed. An exception is thrown if a unit conversion is not possible.

All units supported by module Quanta can be used. Note that the units are case sensitive. Most common units use lowercase characters. A unit can be preceeded by a scaling prefix (like `k` for kilo). In glish one can use `dq.map()` to see the available units and prefixes. Compound units are created when multiplying or dividing values with units.

Units can be given as follows:

- If a column has a unit defined in column keyword `QuantumUnits` or `UNIT`, it automatically gets that unit.
- A constant can immediately be followed by a simple unit.. E.g. `2deg`.
- The result of several expressions have an implicit unit.
  Constants given as positions are in radians (rad).
  Difference of 2 dates is in days (d).
  Inverse trigonometric functions like `asin` give radians.
- When combining values with different units in e.g. an interval, a set, an addition, or a function like `min`, the values are converted to the unit of the first operand or argument with a unit. Values without a unit have by default the unit of the first operand or argument with a unit.

```
3mm-7cm          result is -67 mm
3+3mm            result is 6 mm
3mm<:<3cm        result is interval <3mm,30mm>
[3,4cm,5]        result is [3cm, 4cm, 5cm]
[5, 7cm, 8mm]    result is [5cm, 7cm, 0.8cm]
```

```
[5, 7mm, 8cm]     result is [5mm, 7mm, 80mm]
max(3mm,2cm)      result is 20 mm
5 'km/h' + 1 'm/s'      is 8.6 km/h
iif(F,3min,30sec)       is 0.5 min
```

- Similarly operands of comparison operators and arguments of comparison functions (like `near`) are converted to the unit of the first operand or argument with a unit.
- The result of a multiplication and division is a compound unit if both operands have a unit. Otherwise it is the unit of the argument with a unit.
  Before TaQL supported units, it was needed to divide the TIME column in a MeasurementSet by 86400 to convert it to days, so it could be compared with a given date/time. So, for backward compatibility, a division of a value with unit `s` by a constant 86400 results in unit `d`.
- The result of functions like `SUMSQR` and `SQRT` is a compound unit if the argument has a unit. Note that `sqrt(2m)` will fail, because the square root of a meter does not exist.
- A (sub)expression can be followed by a simple or compound unit. If the subexpression has no unit, it gets the given unit. Otherwise the resulting value is converted to the unit.
  Note that some units can be the same as a reserved word (e.g. `as` or `in`). In that case it has to be escaped or enclosed in quotes.

```
COL \in           set/convert column COL to inch
3 cm              result is 3 cm
3 'km/s'          result is 3 km/second
3mm cm            result is 0.3 cm
(3mm cm)m         result is 0.003 m
(3+3) cm          result is 6 cm
(3+3mm) cm        result is 0.6 cm
[3,4,5]mm         result is [3mm, 4mm, 5mm]
[3,4cm,5]mm       result is [30mm, 40mm, 50mm]
    Note: all values in the set first get the same unit cm
asin(1)           result is pi/2 radians
asin(1) deg       result is 90 degrees
(3mm+7cm) m       result is 0.073 m
```

- If a function argument is expected in a certain unit, values are converted as needed. For example, arguments to functions `sin` and `anycone` are automatically converted to radians.
- When adding or subtracting a value from a date, that value is converted to unit `d` (days).

Units will probably mostly be used in an expression in the WHERE clause or in a CALC command. However, it is also possible to use a unit in the selection of a column in the SELECT clause. For example:
`select TIME d as TIMED from my.ms` In such a case the selection is an expression and the unit is stored in the column keywords. Thus in this example, TIME is stored in a column `TIMED` with keyword `QuantumUnits=d` and the values are converted to days.

4.10 Functions

More than 100 functions exist to operate on scalar and/or array values. Some functions have two names. One name is the CASA/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.
Furthermore it is possible to have user defined functions that are dynamically loaded from a shared library. In section Writing user defined functions it is explained how to write user defined functions. A set of standard UDFs exists dealing with Measure conversions, for example to convert J2000 to apparent. Another set of UDFs deals with measures in MeasurementSets and Calibration Tables.

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.

4.10.1 String functions

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

`integer STRLENGTH(string), integer 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 CAPITALIZE(string)`
: Capitalize a string (make first letter uppercase).

`string LTRIM(string)`
: Removes leading whitespace.

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

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

`string SUBSTR(string, integer ST, integer N)`
: Returns a substring starting at the 0-based position `ST` with a length of at most `N` characters. If the string argument is an array of strings, an array with the substring of each string is returned. The arguments N and ST have to be scalar values. They will be set to 0 if negative.

`string REPLACE(string SRC, PATTERN, string REPL)`
: Replaces all occurrences of PATTERN in SRC by REPL and returns the result. REPL can be omitted and defaults to the empty string. If the first argument is an array of strings, each element in the array is replaced. The arguments PATTERN and REPL have to be scalar values. PATTERN can be a string or a regular expression (see below). For example:
`REPLACE("abcdab", "ab")` results in `cd`
`REPLACE("abcdab", REGEX("^ab"), "xyz")` results in `xyzcdab`

4.10.2 Regex functions

Apart from using regex/pattern constants, it is possible to use functions to form a regex or pattern. 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.

A regex formed this way can only be used in a comparison == or !=. E.g.
`object == pattern('3C*')`
to find all 3C objects in a catalogue.

A few remarks:

1. The regex/pattern functions and operator LIKE work on any string, thus they can be used with any string expression.

2. A Regex is case sensitive. One should use function `upcase` or `downcase` on the string to test to make it case insensitive or use the *i* qualifier on a regex constant.

3. Usually a regex/pattern must match the full string, thus not part of it. However, one can use the `m//` regex constant to do partial matching. Thus something like `m/xx/` matches all strings containing `xx`. Of course, `regex('.*xx.*')` can also be used. In this way the `m//` regex works the same as in languages like Perl, Python, and Glish.

4.10.3 Date/time functions

These functions make it possible to handle dates/times and can be used on a scalar or an array argument. The syntax of a date/time string or constant is explained in section 4.3.7.

`DateTime DATETIME(string)`
: Parse the string and convert it to a DateTime value.

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

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

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

`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`

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

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

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

`integer WEEK(DateTime)`
Get the week number in the year (0 ... 53).
Note that week 1 is the week containing Jan 4th.

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

`string CDATETIME(DateTime), string CTOD(DateTime)`
Get the DateTime as a string like YYYY/MM/DD/HH:MM:SS.SSS.

`string CDATE(DateTime)`
Get the date part of a DateTime as a string like DD-MMM-YYYY.

`string CTIME(DateTime)`
Get the time part of a DateTime as a string like HH:MM:SS.SSS.

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

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

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., the function DATETIME is used implicitly).

4.10.4 Pretty printing functions

Angles (scalar or array) can be returned as strings in HMS and/or DMS format. Currently, they are always formatted with 3 decimals in the seconds.

`string HMS(real)`
Return angle(s) like 12h34m56.789

`string DMS(real)`
Return angle(s) like 12d34m56.789

`string HDMS(realarray)`
: Return angles like 12h34m56.789 (even elements) and 12d34m56.789 (odd elements). It is useful for arrays containing RA,DEC values.

The functions mentioned above and the date/time functions in the previous subsection can format a value in a predefined way only.
The `string` (shorthand `str`) function makes it possible to convert values to strings using an optional format string or width.precision value. It also makes it possible to format dates, times, and angles in a variety of ways.

`string STRING(value, [format]), string STR(value, [format])`

The value can be of any type (except Regex) and can be a scalar or array. The optional format must be a scalar string or numeric value. If no format is given, an appropriate default format will be used.

- A numeric format value defines the width and/or precision. For example:

  ```
  8        defines width 8 and default precision
  20.12    defines width 20 and precision 12
  .8       defines precision 8 and default width
  ```

  In this way precision represents all digits, not only the ones behind the decimal point. A default width or precision is used if not given.

- A string format value can contain a `printf`-style format string, which must include the %-sign. Note that the real and imaginary part of a complex value are formatted separately, so such a format string needs to contain a format specifier for both parts. See printf reference for possible format specifiers. For example:

  ```
  %10d     decimal with width 10
  %010d    decimal with width 10 and filled with zeroes
  %f+%fi   to format a complex value as a+bi
  ```

  Apart from a `printf`-style format string, it is also possible to define a string to format date/time and angle values (which are automatically converted to radians if containing units). Such a format string contains one or more format values as defined in class MVTime. A vertical bar (with optional whitespace) must be used as separator. A string part can be a numeric value defining the precision of the time/angle. For example:

  ```
  YMD              format as YYYY/MM/DD/HH:MM:SS
  DMY|NO_TIME      format as DD-MMM-YYYY
  DMY | DAY | 8    format as Thu-DD-MMM-YYY/HH:MM:SS.SS
  TIME             format a date or angle as HH:MM:SS
  ANGLE|9          format an angle as DD.MM.SS.SSS
  ```

  If such a format string contains an invalid part, it is assumed that the entire string is a `printf`-style format string.

4.10.5 Comparison functions

The exact comparison of floating point values is quite tricky. Two functions make it possible to compare 2 double or complex values with 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).

`bool ISINF(numeric val)`
: Tests if a numeric value is infinite (positive or negative).

`bool ISFINITE(numeric val)`
: Tests if a numeric value is a finite number (not NaN or infinite).

4.10.6 Mathematical functions

Standard mathematical can be used on scalar and array arguments (and a mix of them).

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

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

`double C()`
: Return the value of the speed of light (with unit m/s).

`dnumeric SIN(numeric)`

`dnumeric SINH(numeric)`

`double ASIN(real)`

`dnumeric COS(numeric)`

`dnumeric COSH(numeric)`

`double ACOS(real)`

`double TAN(real)`

`double TANH(real)`

`double ATAN(real)`

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

`dnumeric EXP(numeric)`

`dnumeric LOG(numeric)`
Natural logarithm.

`dnumeric LOG10(numeric)`

`dnumeric POW(numeric, numeric)`
The same as operator `**`.

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

`dnumeric SQRT(numeric)`

`complex COMPLEX(real, real)`

`dnumeric CONJ(numeric)`

`double REAL(numeric)`
Real part of a complex number. Returns argument if real.

`double IMAG(numeric)`
Imaginary part of a complex number. Returns 0 if argument is real.

`real NORM(numeric)`

`real ABS(numeric), real AMPLITUDE(numeric)`

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

`numeric MIN(numeric, numeric)`

`numeric MAX(numeric, numeric)`

`real SIGN(real)`
Return -1 for a negative value, 0 for zero, 1 for a positive value.

`real ROUND(real)`
Return the rounded value of the number. Negative numbers are rounded in an absolute way. E.g. `ROUND(-1.6) = -2.`.

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

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

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

Note that the trigonometric functions need their arguments in radians.

4.10.7 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?

`integer NTRUE(bool)`
: Return number of true elements.

`integer NFALSE(bool)`
: Return 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 elements.

`real MIN(real)`
: Return minimum of all elements.

`real MAX(real)`
: Return maximum of all elements.

`double MEAN(real)`
: Return mean of all elements.

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

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

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

`double RMS(real)`
Return root-mean-squares. (the square root of the sum of
`(a(i)**2)/nelements(a)`.

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

4.10.8 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.
It is important to note that the interpretation of the axes numbers depends on the style being used. E.g. when using glish style, axes numbers are 1-based and in Fortran order, thus axis 1 is the most rapidly varying axis. When using python style, axis 0 is the most slowly varying axis.
Axes numbers exceeding the dimensionality of the array are ignored. 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 section. 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?

`integer NTRUES(bool)`
Return number of true elements.

`integer NFALSES(bool)`
Return 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.

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

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

`double MEANS(real)`
Return mean of elements.

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

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

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

`double RMSS(real)`
Return root-mean-squares. (the square root of the sum of
`(a(i)**2)/nelements(a)`.

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

4.10.9 Array downsampling functions

These functions are a generalization of the functions in the previous section. They downsample an array by taking, say, the mean of every n*m elements. The functions in the previous section downsample by taking the mean of a full line or plane, etc. The most useful one is probably calculating the boxed mean, but the other ones can be used similarly. The width of each window axis has to be given. Missing axes default to 1. Similarly to the partial reduction functions described above, the axes can be given as scalars or as a set.
For example, `BOXEDMEAN(array,3,3)` calculates the mean in each 3x3 box. At the end of an axis the box used will be smaller if it does not fit integrally.
The functions can only be used for arrays, thus not for scalars.

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

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

`double BOXEDMIN(real)`
Return minimum of elements.

`double BOXEDMAX(real)`
Return maximum of elements.

`double BOXEDMEAN(real)`
Return mean of elements.

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

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

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

`double BOXEDRMS(real)`
Return root-mean-squares. (the square root of the sum of
`(a(i)**2)/nelements(a)`.

`double BOXEDMEDIAN(real)`
Return median (the middle element).

4.10.10 Array functions operating in running windows

These functions transform an array into an array with the same shape by operating on a rectangular window around each array element. The most useful one is probably calculating the running median, but the other ones can be used similarly. The half-width of each window axis has to be given; the full width is `2*halfwidth + 1`. Missing axes default to a half-width of 0. Similarly to the partial reduction functions described above, the axes can be given as scalars or as a set.
For example, `RUNNINGMEDIAN(array,1,1)` calculates the median in a 3x3 box around each array element. See the examples how it is applied to an image.
In the result the edge elements (i.e. the elements where no full window can be applied) are set to 0 (or false).
The functions can only be used for arrays, thus not for scalars.

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

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

`double RUNNINGMIN(real)`
Return minimum of elements.

`double RUNNINGMAX(real)`
Return maximum of elements.

`double RUNNINGMEAN(real)`
Return mean of elements.

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

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

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

`double RUNNINGRMS(real)`
Return root-mean-squares. (the square root of the sum of
`(a(i)**2)/nelements(a)`.

`double RUNNINGMEDIAN(real)`
Return median (the middle element).

4.10.11 Type conversion functions

Explicit type conversions can be done using one of the functions below. They can operate on scalars and arrays.

`integer INT(numeric or bool or string)`
Convert the argument to an integer. A real number is truncated (-10.9 results in -10). For a complex number the truncated real part is taken. A bool is converted to 0 (False) or 1 (True). It does not check if a string represent a valid integer. It is interpreted until the first non-valid character, so a string containing a floating point value is truncated.

`double REAL(numeric or bool or string)`
Convert the argument to a real number. For a complex number the real part is taken. A bool is converted to 0 (False) or 1 (True). It does not check if a string represent a valid floating point value. A string is interpreted until the first non-valid character.

`complex COMPLEX(real,real)`
Form a complex number from the given real and imaginary part.

`complex COMPLEX(string)`
Convert the string to a complex number. The number can be given like `(1,2)` or `1+2i`. In fact, any separator (except whitespace) between real and imaginary part is possible. It does

not check if a string represent a valid complex value. The string is interpreted until the first non-valid character, so the last character can be any character (e.g., also `j`).

`bool BOOL(anytype)`
Convert the value to a bool. A numeric type (or date) results in False if the value is 0, otherwise True. A string is case-insensitive. False, F, No, N, -, or 0 results in False, otherwise True.

4.10.12 Array creation functions

`anytypearray ARRAY(anytype,shape)`
This function creates an unmasked 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.
Note that a masked array can be created from an (unmasked) array and a mask using the brackets operator like `ARRAY[MASK]`.

`anytypearray TRANSPOSE(anytypearray[,axes])`
This function transposes an N-dim array. If no axes are given, the array is fully transposed (thus all axes are reversed). Axes can be specified meaning that those axes will become the first axes in the output array. Non-given axes follow thereafter in their natural order.
A possible mask is transposed as well.

`anytypearray DIAGONAL(anytypearray[,firstaxis[,diag]])`
This function takes the diagonal of 2-dim subarrays in an N-dim array resulting in an array with 1 dimension less. For a 2-dim array, it is simply the diagonal of the matrix. For a higher dimensional array, it takes the diagonal of each matrix defined by `firstaxis` and `firstaxis+1`. E.g. in a 3-dim array the diagonals of each XY-plane can be taken. The default for firstaxis is 0.
The `diag` argument tells which diagonal has to be taken. The default 0 means the main diagonal. A value ¡0 means below the main diagonal, while ¿0 means above the main diagonal.

`anytypearray ARRAYDATA(anytype)`
This function returns the array without a mask, thus removes the mask. If the operand is a scalar, it returns a 1-dim array with 1 element.

`boolarray ARRAYMASK(anytype), boolarray MASK(anytype)`
This function returns the mask of an array. If the array has no mask, it returns an empty array. If the operand is a scalar, it returns an empty array.

`anytypearray FLATTEN(anytype)`
This function flattens an N-dim array to a 1-dim array keeping the unmasked elements only. If the operand is a scalar, it returns a 1-dim array with 1 element.

4.10.13 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.

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

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

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

`integer ROWNUMBER(), integer ROWNR()`
Return the row number being tested (first row is row number 0 or 1 depending on the style used).
In combination with function RAND it can, for instance, be used to select arbitrary rows from a table.

`integer ROWID()`
Return the row number in the original table. This is especially useful for returning the result of a selection of a subtable of a Casacore measurement set (see also subqueries in section 4.11 and examples in section 9).

`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.

`double ANGDIST(arg1,arg2)`, `double ANGULARDISTANCE(arg1,arg2)`
Return the angular distance (in radians) between the positions in `arg1` and `arg2`. Both arguments have to be numeric arrays containing an even number of values. Two subsequent values give the RA and DEC (or longitude and latitude) of positions on a sphere. The result is a 1-dim array containing the angular distance between corresponding positions in `arg1` and `arg2`. If either array contains only one position, the result is the distance between that position and each position in the other array. If both arguments contain only 2 values, the result is a scalar. For example:
`angdist(PHASE_DIR[0,], [12h13m45,4d21m39.4, 12h13m49,10d8m4])`
returns an array with shape [2] containing the angular distance between the phase center of the field and the two positions given.

`double ANGDISTX(arg1,arg2)`, `double ANGULARDISTANCEX(arg1,arg2)`
Same as above, but the result is a 2-dim array giving the distance between each position in the first argument and each position in the second argument. Only if both arguments contain a single position, the result is a scalar.

`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`.

If one of the input arrays has a mask, the output array will also have a mask. Each output mask element value is the logical OR of the condition mask element value and the mask value of the element taken from arg1 or arg2.

4.10.14 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 position 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 of 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 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. Where argument name `SOURCE` is mentioned below, only a single source can be used, otherwise multiple sources.

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 the user will specify it as constants.

For example:

`[12h13m54, -5.3.34, 0d1m]` for a single cone.

`[12h13m54, -5.3.34, 0d1m, 1h2m3, 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:
`[12h13m54, -5.3.34, 1h2m3, 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 at least 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.

`integer FINDCONE(SOURCES,CONES)`
Return the 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.

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

`bool CONES(SOURCES,CONES)`
Return a 2-dim bool array. The length of the most rapidly varying axis is the number of cones. The length of the other axis is the number of sources. When using glish style, element `(i,j)` in the resulting 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 most rapidly varying axis, cones as the next axis, and sources as the slowest axis.

Please note that `ANYCONE(SOURCE,CONES)` does the same as `any(CONES(SOURCE,CONES))`, but is faster because it stops as soon as a cone is found.
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 CONES([RA,DEC],
           [SELECT FROM table.cat GIVING [RA,DEC]], 0d0m10)
    FROM table.cat
```

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 (in glish style) `[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 can be done with multiple radii, for example also with 1 arcsecond and 1 arcminute.

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

resulting in an array with glish shape `[3,nrow,1,nrow]`. In this way one can get a better indication how close sources are to the cone centers.

4.10.15 User defined functions

User defined functions (UDF) have to exist in a dynamically loadable library. In TaQL the name of a UDF consists of the name of the library (without lib prefix and extension) followed by a dot and the function name. For example:

```
derivedmscal.pa1
```

denotes function `pa1` in shared library `libderivedmscal.so` or `libcasa_derivedmscal.so`. On OS-X the extension `.dylib` is used.
The library and function name are case-insensitive.

In section Writing user defined functions it is explained how to write user defined functions.

4.10.16 Special MeasurementSet functions

The casacore package comes with several predefined UDFs in library `libcasa_derivedmscal`. The functions can be used on a MeasurementSet or a CASA calibration table (both old and new format). All angles are returned in radians. For calibration tables, where a row contains a single antenna, functions like PA1 are the same as PA2.
For ease of use it is possible to use the alias `mscal`.

`double MSCAL.HA()`
: gives the hourangle of the array center (observatory position).

`double MSCAL.HA1()`
: gives the hourangle of ANTENNA1.

`double MSCAL.HA2()`
: gives the hourangle of ANTENNA2.

`double MSCAL.HADEC()`
: gives the hourangle/declination of the array center (observatory position).

`double MSCAL.HADEC1()`
: gives the hourangle/declination of ANTENNA1.

`double MSCAL.HADEC2()`
: gives the hourangle/declination of ANTENNA2.

`double MSCAL.LAST()`
: gives the local sidereal time of the array center.

`double MSCAL.LAST1()`
gives the local sidereal time of ANTENNA1.

`double MSCAL.LAST2()`
gives the local sidereal time of ANTENNA2.

`double MSCAL.PA1()`
gives the parallactic angle of ANTENNA1.

`double MSCAL.PA2()`
gives the parallactic angle of ANTENNA2.

`doublearray MSCAL.AZEL1()`
gives the azimuth/elevation of ANTENNA1.

`doublearray MSCAL.AZEL2()`
gives the azimuth/elevation of ANTENNA2.

`doublearray MSCAL.UVW()`
gives the UVW coordinates in J2000 (in meters).

By default all these functions will use the direction given in column PHASE_DIR of the FIELD subtable. It is possible to use another column by giving its name as a string argument (e.g., 'HA(DELAY_DIR)').
It is also possible to use an explicit direction which must be given as [RA,DEC] in J2000 or as a case-insensitive name of a planetary object (as defined by the casacore Measures). For example:

```
derivedmscal.azel1([5h23m32.76, 10d15m56.49])
derivedmscal.azel1('MOON')
```

These examples give the azimuth and elevation of the given directions for each selected row in the MeasurementSet, thus using the position of ANTENNA1 and the times in these rows.

If a string value is given, it is first tried as a planetary object. Theoretically it is possible that a column has the same name as a planetary object. In such a case the name can be escaped by a backslash to indicate that a column name is meant. For example:

```
derivedmscal.azel1('\SUN')
```

means that column SUN in the FIELD table has to be used.

The `STOKES` function makes it possible to convert the Stokes parameters of a DATA column in a MeasurementSet, for instance from linear or circular to iquv. It is also possible to convert the weights or flags, i.e., to combine them in the same way as the data would be combined.

`complexarray MSCAL.STOKES(complexarray, string)`
converts the data.

`doublearray MSCAL.STOKES(doublearray, string)`
combines the weights.

`boolarray MSCAL.STOKES(boolarray, string)`
combines the flags.

In all cases the case-insensitive string argument defines the output Stokes axes. It must be a comma separated list of Stokes names. All values defined in the casacore class `Stokes` are possible. Most important are:

- XX, XY, YX, and/or YY.
  LINEAR or LIN means XX,XY,YX,YY.
- RR, RL, LR, and/or LL.
  CIRCULAR or CIRC means RR,RL,LR,LL.
- I, Q, U, and/or V.
  IQUV or STOKES means I,Q,U,V.
- PTOTAL is the polarized intensity (sqrt(Q**2+U**2+V**2))
- PLINEAR is the linearly polarized intensity (sqrt(Q**2+U**2))
- PFTOTAL is the polarization fraction (Ptotal/I)
- PFLINEAR is the linear polarization fraction (Plinear/I)
- PANGLE is the linear polarization angle (0.5*arctan(U/Q)) (in radians)

If not given, the string argument defaults to 'IQUV'. For example:

```
select mscal.stokes(DATA,'circ') as CIRCDATA from my.ms
```

creates a table with column CIRCDATA containing the circular polarization data.

The `BASELINE` function makes it possible to do selection on baselines in a MeasurementSet or CalTable using the special CASA baseline selection syntax

```
select from my.ms where mscal.baseline('RT[2-4]')
```

selects the baselines containing an antenna whose name matches the pattern in the function argument. Note that autocorrelations are not selected this way.

Similar functions `TIME, FIELD, SCAN, SPW, UVDIST, STATE, OBS`, and `ARRAY` use the CASA selection syntax to select on the corresponding fields. The aforementioned document also describes these syntaxes. For example:

```
select from my.ms where mscal.field('1~4')
```

4.10.17 Special Measures functions

These functions make it possible to convert measures like directions, epochs, and positions from one reference frame to another. All conversions supported by casacore's Measures are possible. For example:

```
meas.galactic (-6h52m36.7, 34d25m56.1, "J2000")
meas.azel ("MOON", date(), "WSRT")
```

The first example converts a J2000 position to galactic coordinates. The second example gives the moon's azimuth/elevation at the WSRT at the current date/time.

The following basic functions are available. Most functions return double angle values with unit rad. Only the RISESET function returns date/time values.
Note that all names used below are case-insensitive.

`doublearray MEAS.DIR(toref, direction, epoch, position)`
converts a direction to the reference type given by the 'toref' string. The epoch and position arguments only need to be given if the conversion needs frame information (e.g. when converting to apparent).
Function name MEAS.DIRECTION can be used as well.

`DateTimearray MEAS.RISESET(direction, epoch, position)`
returns the rise and set date/times (UTC) of the sources given in the direction argument for the given dates and positions. Note that the TIME or CTIME function can be used on the result to get the time part only (as double or string).

`doublearray MEAS.EPOCH(toref, epoch, position)`
converts an epoch to the reference type given by the 'toref' string. The position argument only needs to be given if the conversion needs frame information.

`doublearray MEAS.POS(toref, position)`
converts a position to the reference type given by the 'toref' string.
Function name MEAS.POSITION can be used as well.

For ease of use several specialized functions are defined with an implicit 'toref' argument.

`doublearray MEAS.J2000(direction, epoch, position)`
converts a direction to J2000.

`doublearray MEAS.B1950(direction, epoch, position)`
converts a direction to B1950.

`doublearray MEAS.APP(direction, epoch, position)`
converts a direction to apparent coordinates.
Function name MEAS.APPARENT can be used as well.

`doublearray MEAS.HADEC(direction, epoch, position)`
converts a direction to hourangle/declination.

`doublearray MEAS.AZEL(direction, epoch, position)`
converts a direction to azimuth/elevation.

`doublearray MEAS.ECL(direction, epoch, position)`
converts a direction to ecliptic coordinates.
Function name MEAS.ECLIPTIC can be used as well.

`doublearray MEAS.GAL(direction, epoch, position)`
converts a direction to galactic coordinates.
Function name MEAS.GALACTIC can be used as well.

`doublearray MEAS.SGAL(direction, epoch, position)`
converts a direction to supergalactic coordinates.
Function name MEAS.SUPERGAL or MEAS.SUPERGALACTIC can be used as well.

`doublearray MEAS.LAST(epoch, position)`
converts an epoch to local sidereal time.
Function name MEAS.LST can be used as well.

`doublearray MEAS.ITRF(toref, position)`
converts a position to ITRF coordinates.

`doublearray MEAS.WGS(toref, position)`
converts a position to WGS84 coordinates.

The name of the last two functions can have a suffix indicating how positions are returned.
- XYZ means as x,y,z
- LL or LONLAT means as lon,lat
- H or HEIGHT means as height It defaults to XYZ.

The function arguments can be given in a variety of ways.

- 'toref' is a constant scalar string giving the reference type to convert to. See the Measure classes MDirection, MEpoch, and MPosition, for an overview of the types.
- 'direction' gives one or more directions to convert. They can be given in several ways.
  - As a constant scalar or array of strings giving one or more planetary objects like MOON or VENUS and/or giving the name of standard sources (CasA, CygA, TauA, VirA, HerA, HydA, or PerA). In the future support for comets might be added.
    The names are case-insensitive.
  - As 2 constant double scalar arguments giving ra and dec (or longitude and latitude).
  - As a double array with an even number of elements giving ra/dec or longitude/latitude of one or more directions. It can be a constant array, but it can also be a column or an expression using a column.

  If a column or a column slice is given, the reference type stored in the column keywords will be recognized. In other cases the reference type should be given in the next string argument. If not given, it defaults to J2000.
  For example:

```
['MOON','sun', 'venus']            # 3 planetary objects
12h23m17.5, 23d56m43.8, 'B1950'    # 2 scalar constants (as B1950)
[12h23m17.5, 23d56m43.8]           # constant array (default J2000)
PHASE_DIR[0,]                      # 1st direction in given column
```

- 'epoch' gives one or more epochs to use. Similar to directions the reference type is taken from the column keywords or can be given in the next argument. It defaults to UTC.
  Epochs can be given in three ways:

  - As a scalar or array containing double values. It can be a constant or a column (expression).
  - As a scalar or array containing DateTime values.
  - As a scalar or array containing String values representing date/time. They will automatically be converted to DateTime values using function `datetime`.

  For example:

```
datetime()                          # current date/time
'today'                             # current date/time
[select unique TIME from my.ms]     # all times from some MS
9Oct2011/12:00:00, 'UTC'            # given UTC time
```

  Note that in the last example 'UTC' is not necessary, because it is the default.

- 'position' gives one or more directions to use. They can be given as x,y,z or as lon,lat with an optional height. Usually the unit of the first value defines if x,y,z or lon,lat is used. It is, however. also possible to use suffices like XYZ or LL in the reference type given in the next argument.

  - As a scalar or array of observatory names to use their positions in the Observatory table.
  - As 2 or 3 constant double scalar values giving xyz, lonlat, or lonlat/height.
  - As a double array giving one or more positions in xyz or lonlat. Similar to directions it can be a column (expression) where the reference type is taken from the column keywords.
  - As two constant double arrays giving lonlat and height of one or more positions. The array sizes have to match (thus the size of the lonlat array must be twice the size of the height array).

  If needed, the reference type (with optional suffix) can be given in the next argument. The reference type defaults to ITRF if xyz is used, otherwise to WGS.
  For example:

```
'WSRT'                              # WSRT position
5deg, 52deg                         # 2 scalar constants (WGS84)
(5deg, 52deg]                       # same, but as array
5deg, 52deg, 5m                     # WGS84 with height
[5deg, 52deg], [5m]                 # same, but as array
3.8288e+06m, 442449, 5.0649e+06     # xyz as scalars (ITRF)
[41.84m, 4.835, 55.722], 'WGS'      # xyz as array (WGS84)
POSITION                            # POSITION column
```

A few more elaborate examples are given below.

```
meas.last (date('15Oct2011/15:34'), 5deg, 52deg)
meas.last (date('15Oct2011/15:34'), 5deg, 52deg) d
```

It calculates the local apparent sidereal time for the given date and position. Note that the first example returns the value in seconds, while the second returns it in days (because unit 'd' was given).

```
meas.azel ("JUPITER", [select unique TIME from ~/GER1.MS],
          ["WSRT","VLA"])
```

It calculates Jupiter's azimuth/elevation for WSRT and VLA for all times returned by the subquery (see next section for subqueries).

```
calc meas.b1950(PHASE_DIR[0,]) from ~/GER1.MS/FIELD'
```

It converts the PHASE_DIR directions in the FIELD table to B1950. Note that no frame information is needed for such a conversion.

4.11 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 [SELECT NAME FROM ::ANTENNA][ANTENNA1]
       FROM ~/GER1.MS
```

This example shows how a subquery is used to join the main table of a MeasurementSet with its ANTENNA subtable. The subquery returns a list with the names of all antennae, which subsequently is indexed with the antenna number to get the antenna name for each row in the main table.

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

This example 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 Casacore 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 section 3.8). 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 Casacore 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 given by 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. Select if any channel has a UV distance ¡ 100 wavelengths.

```
select from MY.MS where any(sqrt(sumsqr(UVW[:2])) / c() *
    [select CHAN_FREQ from ::SPECTRAL_WINDOW][DATA_DESC_ID,]
       < 100)
```

   In a MeasurementSet the UVW coordinates are stored in meters, so they have to be multiplied with the frequency and divided by speed of light to get them in wavelengths.
   Because TaQL has no proper join operation, it is not possible to select directly on the DATA_DESC_ID. However, using a nested query and indexing the result with the DATA_DESC_ID has the same effect. It only requires that CHAN_FREQ has the same length in all rows in the subtable.

6. Calculate the angular distance between the sun and the moon as seen from the WSRT for the coming 30 days.

```
calc angdist(meas.app('SUN',  date()+[0:31], 'WSRT'),
             meas.app('MOON', date()+[0:31], 'WSRT'))
```

5 Some further remarks

5.1 Joining tables

As discussed in some previous sections it is possible to join tables on row number. Two examples show how to do it.

```
SELECT FROM mytable t1,othertable t2
  WHERE not all(near(t1.data, t2.data))
```

This command can be used to check if the data in mytable is about equal to the data in othertable. Both tables have to have the same number of rows.
The join is done on row number, thus the data of the same rows are compared.

```
SELECT [SELECT NAME FROM ::ANTENNA][ANTENNA1]
       FROM ~/GER1.MS
```

This example shows how a subquery is used to join the main table of a MeasurementSet with its ANTENNA subtable. The subquery returns a list with the names of all antennae, which subsequently is indexed with the antenna number to get the antenna name for each row in the main table.
The join is done using the ANTENNA1 column which gives the row number in the subtable, thus the index in the subquery result.

5.2 Optimization

A lot of development work can be done to improve the query optimization. At this stage only a few simple 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 reals 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.

TaQL does not recognize common subexpressions nor does it attempt to optimize the query. It means that 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 section 4.8 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 all or some rows in the first table of the table list. Other tables can be used in clauses like SET and WHERE.
`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.

```
UPDATE this.ms, that.ms t2 set DATA=t2.DATA, FLAG=t2.FLAG
                    where all(FLAG)
```

to copy the DATA and FLAG column of that.ms to this.ms for rows where all data are flagged. Note the use of the shorthand (alias) `t2`.

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
UPDATE vla.ms SET FLAG=F
```

The first example sets only a single array element, while the second one sets an entire row in the array. The second and third example show 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 (the result will be truncated).
Unit conversion will be done as needed. Thus if a column and its expression have different units, the expression result is automatically converted to the column's unit. Of course, the units must be of the same type to be able to convert the data.

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.

Thus to swap the values of the ANTENNA1 and ANTENNA2 column, one can **not** do:

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

TaQL does not have temporary variables to solve the problem. However, the following could be used:

```
UPDATE vla.ms SET ANTENNA1=ANTENNA1+ANTENNA2,
                  ANTENNA2=ANTENNA1-ANTENNA2,  #A2 is old A1
                  ANTENNA1=ANTENNA1-ANTENNA2   #A1 is old A2
```

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 and units of expressions and columns have to conform in the same way as for the UPDATE command; values have to be convertible to the column data type and unit.
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 column-list 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,...], UNIT='s',
                     DMTYPE='s',DMGROUP='s',COMMENT='s']
```

The possible data type strings are given in section 4.1. 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.
- `UNIT='s'` defines the unit to be used for the column.
  It can be any valid unit (simple or compound). It is a string, thus must always be enclosed in quotes.
- `COMMENT` defines comments for the column.
  It has a string value, thus quotes have to be used.
- `DMTYPE, DMGROUP` are rather specific and are for the expert user.
  They have a string value, thus quotes have to be used.

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 Python or 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. If shapes have to be given here, they always have to be in casacore format, thus in Fortran order. TaQL has no knowledge about these internals.
`COLUMNS` is a list of column names defining all columns that have to be bound to the data manager.

8 Counting in a table

TaQL does not have a GROUPBY command (yet). However, it has a command that can be used to count the number of occurrences in a table.

The exact syntax is:

```
COUNT column-list FROM table-list [WHERE expression]
```

It counts the number of rows for each unique tuple in the column list of the table (after the possible WHERE selection is done). For example:

```
COUNT TIME FROM my.ms
```

counts the number of rows per timestamp.

```
COUNT ANTENNA1,ANTENNA2 FROM my.ms
```

counts the number of rows per baseline.

As in the other TaQL commands a column in the column list can be any expression, but that will be slower than straight columns.

9 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 expression [FROM table_list]
```

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.
The following syntax is still available for backward compatibility:

```
CALC FROM table_list CALC expression
```

```
CALC 1in cm
```

is a simple expression not using a table. It shows how the CALC command can be used as a desk calculator to convert 1 inch to cm.

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

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.

10 Examples

10.1 Selection examples

Some examples are given starting with simple ones.

10.1.1 Reference table results

The result of the following queries is a reference table, because no expressions have been given in the column-list. This will be the most common case when using TaQL.

`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 rows if there happen to be several equal values.

```
SELECT FROM some.MS WHERE
   near(MJD(1999/03/30/17:27:15), TIME)
```
selects the rows with the given time from a MeasurementSet.
Note that the TIME is stored in seconds, but will automatically be converted to days.

```
SELECT FROM some.MS where TIME 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 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 this.ms t1, that.ms t2
   WHERE !all(near(t1.DATA, t2.DATA, 1e-5))
```
selects all rows where the DATA columns in both tables are not equal (with some tolerance).
Note the use of shorthands t1 and t2.

```
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.

```
select findcone(REFERENCE_DIR[0,],
   [16h34m33.805,62d45m36.83, 12h29m06.7,2d3m9], 1arcsec)
   from MS/FIELD //compares the direction given in the first argument with the directions
```
given in the second function argument using the search radius given in the third argument. It returns the index of the first matching cone (thus 0 or 1). If no cone matches, it returns -1.
It can be used in the following example to find the name of the source matching a direction.

```
select ['unknown','3C343', '3C273'][1+int(findcone(...))]
```
from MS/FIELD where ... is the findcone argument list given in the previous example. 1 is added to cope with the case that no cone matches.
Note that the conversion to int is done to circumvent a small bug in TaQL.

10.1.2 Plain table results

The following examples result in a plain table, thus in a deep copy of the query results, because the column-list contains an expression or a data type.

`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 a 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,0) AS DATA_MEAN C4 FROM my.ms`
creates a table of 1 column with name `DATA_MEAN`. Column DATA in a Casacore 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).
Note that this command is valid when using python style; in glish style `MEANS(DATA,2)` should be used.

10.2 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.

10.2.1 Applying running median to an image

The following piece of Python code shows how a running median can be applied to an Casacore image.

```
t1 = tablecommand ('update my.imgd set map =
                        map - runningmedian(map,25,25)')
```

First the image is copied and thereafter the running medians are subtracted from the data in the copy. It uses a half window size of 25x25, thus the full window is 51x51.
When doing this, one should take care that in case of a spectral line cube the image is not too large, otherwise it won't fit in memory. If too large, it should be done in chunks like:

```
t1 = tablecommand ('update my.imgd set map[,,sc:ec,] =
                        map[,,sc:ec,] -
                        runningmedian(map[,,sc:ec,],25,25)')
```

where `sc` and `ec` are the start and end frequency channel. In this example it is assumed that the axes of the image are RA, DEC, freq, Stokes.
Note that `imagecalc` is used to copy the image. It might also be done with table functions, but `imagecalc` has the great advantage that it also works on FITS and Miriad images.

10.3 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.

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

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,0)))]])
```

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 first axis, thus the frequency axis.

11 Interface to TaQL

User and a programmer interfaces to TaQL are available. The program `taql` and some Python and Glish functions form the user interface, while C++ classes and functions form the programmer interface.

11.1 Python interface `pyrap`

The main TaQL interface in Python is formed by the `query` function in module `table`. 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.

The functions `sort` and `select` exist as convenience functions for a query consisting of a sort or column selection only. Both functions have an optional second `name` parameter to make the result persistent.

```
t1 = tab.sort ('time')
t1 = tab.select ('column1,column2')
```

The `calc` function can be used to execute a TaQL `calc` command on the current table. The result can be kept in a variable. For example, the following returns a vector containing the median of the `DATA` column in each table row:

```
med = t.calc ('median(DATA)')
```

It is possible to embed Python 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

```
from pyrap.tables import *
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 pyrap.util.

The other function that can be used is taql (or its synonym `tablecommand`). The full TaQL command has to be given to that command. The result is a table object. E.g.

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

By default, these commands will use the Python style for a TaQL statement. The `style` argument can be used to choose another style.

11.2 Interface to Glish

The Glish interface is formed by script `table.g`. By default, it will use the Glish style for a TaQL statement. For example:

```
include 'table.g'
tab := table('mytable')
seltab1 := tab.query ('column1 > 0')
seltab2 := seltab1.query (query='column2>5',
                          sortlist='time',
                          columns='column1,column2',
                          name='result.tab')
t := tablecommand('select from GER.MS where ANTENNA1==1',
                  style='');    # use default (glish) style
med := t.calc ('median(DATA)')
```

11.3 Program `taql`

The program `taql` makes it possible to execute TaQL commands from the shell.
The first way to run it is by giving the TaQL command as a single (quoted) argument. It will

execute the command and exit.
The other way is to run it interactively by giving no command argument. It will run until the user stops via the command `exit`, `quit`, or `q` or by giving ctrl/D. A few types of commands can be given:

- `help`, `h` or `?` shows brief help information.
- A full TaQL command. If no GIVING part is given, the resulting table is not kept. It shows the number of matching rows.
- A full TaQL command preceeded by `varname=`, where `varname` is the name under which the resulting table is kept in this session. Thus the result is not a persistent table (unless GIVING was given), but it is kept temporarily. The name can be used in subsequent commands like `SELECT FROM $varname`.
- `varname=` without a further command removes the temporary result.
- `varname` shows the number of rows in the temporary result. It can be followed by one or more question marks to show the column names and details about them.
  Note that if an unknown varname is given, it is treated as a TaQL command resulting in a parse error.

If columns are selected in the TaQL command, their values are printed. Epoch, position, and direction measures are printed in a formatted way. Also the result of CALC commands is printed. Otherwise only the number of selected (or updated or deleted) rows is printed.

11.4 C++ interface

The C++ programmer can use TaQL commands and expressions at various levels,

11.4.1 TaQL query string

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.
There is no style argument, so if an explicit style is needed it should be the first part of the TaQL statement. Note that no style defaults to the Glish style.

11.4.2 Expression string

The function `parse` in RecordGram.h can be used to parse a TaQL expression. The result is a TableExprNode object that can be evaluated for each row in the table. E.g.

```
Table tab("mytable");
TableExprNode expr = RecordGram::parse (tab, "column1>0");
Table seltab1 = tab(expr);
```

The example above does the same as the first example in the previous section. There are, however, better ways to use this functionality.

```
Table tab("somename");
TableExprNode expr = RecordGram::parse (tab, "ANTENNA1=1");
for (uInt row=0; row<tab.nrow(); ++row) {
  if (expr.getBool(row)) {
    // expression is true for this row, so do something ...
  }
}
```

The example above shows a boolean scalar expression, but it can also be a numeric expression or an array expression as shown in the example below. Note that TaQL expression results have data type Bool, Int64, Double, DComplex, String, or MVTime.

```
TableExprNode expr = RecordGram::parse (tab, "abs(DATA)");
Array<Double> data;
for (uInt row=0; row<tab.nrow(); ++row) {
  expr.get (row, data);
}
```

11.4.3 Expression classes

The other expression interface is a true C++ interface having the advantage that C++ variables can be used directly. 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.

12 Writing user defined functions

A user defined function has to be written as a class in C++ derived from the abstract base class UDFBase. The documentation of this base class describes how to write a UDF. Furthermore one can look at class UDFMSCal that contains the UDFs described in subsection User defined functions.

Note that a class can contain multiple UDFs as done in UDFMSCal. Also note that a single UDF can operate on multiple data types which is similar to a function like `min` that can operate on scalars and arrays of different data types.

TaQL finds a UDF by looking in a dictionary that maps the UDF name to a function constructing an object of the UDF class. If not found, it tries to load the shared library with the lowercase name of the library part of the UDF (like in `derivedmscal.pa1`). If the load is successful, it calls an initialization function in the shared library that should add all UDF functions in the library to the dictionary. The description of the UDFBase class shows how this should be done.

13 Possible future developments

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

- Implement the GROUP BY and HAVING clause, possibly with ROLLUP and CUBE.
- Add explicit JOIN clause (probably only equi-joins).
- Add UNION, INTERSECTION, and DIFFERENCE.
- Add masked arrays.
- 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.
- Optimize IN if righthand is an integer bounded set by turning it into a boolean vector.