NOTE 199 – Table Query Language

Ger van Diepen, ASTRON Dwingeloo

2016 Apr 4

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. User defined functions can be added easily. It also has full support of grouping/aggregation and nested queries. An operation that can be expressed in a single function is the matching of two sky catalogues.
It can be used from C++, Python, and the Casacore program `taql`.

1.0	1997 Feb 9	Original version
2.0	2010 Nov 5	UPDATE, INSERT, DELETE and COUNT commands
3.0	2015 Jul 29	GROUPBY and HAVING clause
3.1	2016 Apr 4	Masked arrays; ALTER TABLE and SHOW commands
3.2	2018 Feb 14	WITH clause
3.3	2018 Jun 10	Meas support for frequency, doppler, radialvelocity, earthmagnetic

Contents

1 Introduction

The Table Query Language (TaQL, rhymes with bagel (though some people pronounce it as tackle)) is a language for querying and manipulating data in Casacore tables. It makes it possible to get information from the data content in the columns and keywords of arbitrary tables. It supports arbitrary complex expressions including units, extended regular expressions, and many functions. User defined functions written in C++ or Python are supported, which is used to support coordinate conversions in TaQL. TaQL also makes sorting and column selection possible. Furthermore, TaQL has commands to update, add or delete rows and columns in a table and to create a new table.

The first sections of this document explain the syntax and show the options. The last sections give several examples and 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.

1.1 TaQL vs SQL

TaQL is modeled after SQL and contains a subset 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 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 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 reduce 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 support of maximum string distance (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).
- Many aggregate functions that can be used with GROUPBY.
- The COUNT command exists to count the occurrences of column values. Although it can still be used, this command is obsolete now GROUPBY is fully supported.
- 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, possibly implemented in Python. Some standard UDFs exist to deal with MeasurementSets and to do measure conversions (for directions, epochs, positions, and stokes).
- In the WITH clause the order of the table and alias is reversed compared to SQL. In this way they have the same order as the table specifications in the FROM clause.

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

2 TaQL Commands

2.1 Command summary

A command can be followed by a semicolon and/or a comment, indicated by a leading hash-sign. They are ignored. For example:

```
calc sin(pi());   # calculate sine of pi
```

The available TaQL commands are shown below. The square brackets are not part of the syntax, but indicate the optional parts of the commands.

- show (or help)

  ```
  SHOW [type ...]
  ```

  can be used to give some TaQL explanation or to show table information. A sole `show` command shows the possible options. HELP is a synonym for SHOW.

- selection

  ```
  SELECT [[DISTINCT] expression_list]
    [INTO table [AS options]]
    [FROM table_list]
    [WHERE expression]
    [GROUPBY expression_list]
    [HAVING expression]
    [ORDERBY [DISTINCT] sort_list]
    [LIMIT expression] [OFFSET expression]
    [GIVING table [AS options] | set]
    [DMINFO datamanagers]
  ```

  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 first table list.

- addition

```
  INSERT INTO table_list SET column=expr, column=expr, ...
or
  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. Although the command can still be used, it is basically obsolete because the same (and more) can be achieved with the GROUPBY clause and aggregate functions in the SELECT command.
Furthermore, usually GROUPBY is faster.

- 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 [AS options]
  [column_spec]
  [LIMIT ...]
  [DMINFO datamanagers]
```

It can be used to create a new table with the given columns and number of rows. Optionally specific table and data manager info can be given.

- table structure modification

```
ALTER TABLE table
  [ADD COLUMN [column_specs] [DMINFO datamanagers]]
  [COPY COLUMN col TO colnew [colattr] [SET [expr]], ...]
  [RENAME COLUMN column_pair_list]
  [DROP COLUMN column_list]
```

```
[SET KEYWORD key=value, key=value, ...]
[COPY KEYWORD key=other, key=other, ...]
[RENAME KEYWORD keyword_pair_list]
[DROP KEYWORD keyword_list]
[ADD ROW nrow]
```

The subcommands of the ALTER TABLE command (shown above) can be used to add, rename, and remove columns and keywords and to add rows. Multiple such subcommands can be given, separated by white space.

All TaQL commands, except SHOW, can be preceded by the clause

```
WITH table_list
```

which can be used to create one or more temporary tables to be used in the subsequent command. Section 3.2 contains a detailed description of this clause.

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 7 (Modifying a table) explains the UPDATE, INSERT, and DELETE commands in more detail. The CREATE TABLE command is explained in section 8 (Creating a table). The ALTER TABLE command is explained in section 9 (Modifying the table structure).
Section 10 (Counting in a table) explains the COUNT command in more detail.
Section 11 (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 preceding 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`

The following values are also possible and are described in the next subsections.

- `synonym=libname` to define a synonym for a user defined library.
- `TRACE` or `NOTRACE` to (un)set tracing.
- `TIME` or `NOTIME` to (un)set timing.

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 them to the style being used. Thus when using python style, the axes will be reversed (data will not be transposed). **Note: unless said differently, all examples in this document are done using the Python style.**

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

2.2.1 UDF library synonyms

The style clause can also be used to define synonyms for the library names of user defined functions. For example:

```
using style mscal=derivedmscal
```

defines the synonym `mscal`. Synonyms make it easier (i.e., less typing) to specify user defined functions. Note that the synonym in the example above is automatically defined by TaQL as well as the synonym `py` for `pytaql`.

2.2.2 Tracing

It is possible to get tracing output during the execution of a TaQL command by using the case-insensitive value `TRACE` in the `using style` command. It can bes useful for debugging purposes.

2.2.3 Timing

It is possible to time a TaQL command by using the case-insensitive value `TIME` in the `using style` command. For historical reasons it is also possible to use the the case-insensitive keyword `TIME` before or after the optional style command.

Timing shows the total execution time and the times needed for various parts of the TaQL command 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).

2.3 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   GROUPBYROLLUP
HAVING
ILIKE          IN        INCONE     INSERT     INTERSECT   INTO
JOIN
LIKE           LIMIT
MINUS
NODUPLICATES   NOT
OFFSET         ON        OR         ORDERBY
SAVETO         SELECT    SET        SUBTABLES
T              TO        TOP        TRUE
UNION          UNIQUE    UPDATE     USINGSTYLE
VALUES
WHERE          WITH
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 that way by escaping it with a backslash like `\AS`. When reading further, the meaning of

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

might become 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 and 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 selected rows in 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 FROM clause can be omitted from the select. In that case no columns can be used in the selection, but functions like `rand` and `rowid` make variable output possible. Clauses like ORDERBY can be given. The GIVING (or INTO) might be useful to store the result in a table.

There is no explicit JOIN clause, but it is possible to equi-join tables on row number. Such tables must 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 6.1.

3.1 SELECT command overview

The SELECT command consists of various clauses of which most are optional. The full command looks as follows where the optional parts are shown in square brackets.

```
[WITH table_list]
SELECT [[DISTINCT] column_list]
  [INTO table [AS options]]
  [FROM table_list]
  [WHERE expression]
  [GROUPBY expression_list]
  [HAVING expression]
  [ORDERBY [DISTINCT] sort_list]
  [LIMIT expression] [OFFSET expression]
  [GIVING table [AS options] | set]
  [DMINFO datamanagers]
```

The clauses are executed in a somewhat different order.

1. WITH to create the temporary tables to be used.
2. FROM to define the tables to be used.
3. WHERE to select the rows.
4. GROUPBY to group selected rows.
5. SELECT to fill select columns used in HAVING or ORDERBY.
6. HAVING to select groups.
7. SELECT to fill the remaining select columns.
8. ORDERBY to sort the result.
9. LIMIT (or TOP) and OFFSET to ignore entries in the sorted result.
10. DMINFO to define the data managers to be used if the result is stored in a plain table.
11. GIVING/INTO to store the final result. stored in a plain table. See section 8.2 how to specify them.

All clauses are explained in full detail in the subsequent sections.

3.1.1 Column/keyword lookup

Expressions in the various clauses will normally use column names to select, sort, or group a table. It is also possible to use table keywords or column keywords by giving their names. Furthermore, it is possible to use a column, created in the SELECT clause, in the HAVING and ORDERBY clauses. This can save time in both specifying and executing the command, because a possibly complicated expression can be used to create such a column. If such columns are used, that part of the SELECT is executed before HAVING.

TaQL uses the following lookup scheme for column/keyword names.

1. If preceded by a shorthand (like in `t0.DATA`), the name is looked up in the corresponding table given in the FROM clause.
2. If not preceded by a shorthand, a name is first looked up in the select columns. If not found, the name is looked up in the first table given in FROM.
3. A name is first looked up as a column in a table. If not found, it is looked up as a table keyword.

See the discussion of column names and keyword names for more details.

3.2 WITH table_list

Sometimes it is useful to create a temporary table to be used thereafter in a SELECT command (or another TaQL command) containing subqueries. It can make the command more clear or it can optimize the command by factoring out subqueries that are used multiple times. For example:

```
with [select ANTENNA1,ANTENNA2,gntrue(FLAG) as NFLAG
      from 3C343.MS
      where ANTENNA1!=ANTENNA2
      groupby ANTENNA1,ANTENNA2] as t1
select STATION,gsum(NFLAG)
from [[select NFLAG,ANTENNA1 as ANTENNA from t1],
      [select NFLAG,ANTENNA2 as ANTENNA from t1]]
groupby ANTENNA orderby ANTENNA]
```

This command counts the number of flagged visibilities per antenna. It looks somewhat complicated and can only be fully understood once the entire TaQL note has been read.
The imporant thing is that the WITH command creates a small temporary table containing the number of flagged visibilities per baseline. It is used twice in the subsequent SELECT command (by concatenation). First for ANTENNA1, thereafter ANTENNA2. The above query command is about twice as fast as something like

```
select ANTENNA,gntrue(FLAG)
from [[select FLAG,ANTENNA1 as ANTENNA from 3C343.MS
       where ANTENNA1!=ANTENNA2],
      [select FLAG,ANTENNA2 as ANTENNA from 3C343.MS
       where ANTENNA1!=ANTENNA2]]
groupby ANTENNA orderby ANTENNA
```

because it processes all flags in the MeasurementSet only once instead of twice.

It is important to note that, unlike the SQL WITH clause, the order is 'WITH table AS alias'. This is the same order as used in the FROM clause.

3.3 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 tables in the table_list (see below). If no column_list is given, all columns of the first table 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 precede a column name with a table shorthand indicating which table in the FROM clause has to be used. If not given, a column will be looked up in the first table. Note that if equally named columns from different tables are used, one has to get a new name, otherwise a 'duplicate name' error will occur. For example:

```
select t0.DATA, t1.DATA as DATA1 from 3C343.MS t0, 3C343_1.MS t1
```

3.3.1 Wildcarded SELECT columns

Apart from giving exact column names, it is also possible to use wildcards by means of a UNIX filename-like pattern (`p/pattern/`) or a regular expression (as `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. An operator has to be given before the pattern or regex. Operator `~` means inclusion of the matching columns. Operator `!~` means exclusion of the matching columns included so far by means of a pattern or regex since the last explicit column name or expression.
A special pattern is * (which is the same as `p/*/`). If `!~` is used at the first pattern or regex, it is assumed that all columns are included (as if * was given before).
The pattern or regex (except *) can be preceded by a table shorthand denoting that the columns have to be taken from that table. For example:

```
select *, !~p/*_DATA/ from 3C343.MS
select !~p/*_DATA/ from 3C343.MS          # * is assumed first
select !~p/t1.*_DATA/ from 3C343.MS t1    # with shorthand t1
```

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 (in the first case because it is explicitly selected).
Note it is not possible to change the name or data type of wildcarded columns.

3.3.2 Expressions in SELECT column_list

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 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.

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
```

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.

Special aggregate functions (e.g., `gmin`) exist to calculate an aggregated value (minimum in this example) per group of rows where the grouping is defined by the GROUPBY clause. The entire column is a single group if no GROUPBY is given. Aggregation is discussed in more detail in section 5.

If a column_list is given and if all columns (and/or expressions) are scalars, the column_list can be preceded 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.

3.3.3 Masked array in column_list

If an expression in the column_list is a masked array, it is possible to create two columns from it: one for the data, one for the mask. This can be done by combining them in parentheses like `(DATA,MASK)`. A possible data type given after the column names only applies to the data column, since the mask column always has data type Bool. For example:

```
select means(DATA[FLAG],0) as (MD,MM) C4 from in.ms giving out.tab
```

The select results in a masked array containing the means along axis 0. Both column MD and MM are filled with the contents of the masked array. MD (with data type C4) contains the means over the first axis of the unmasked elements; MM contains the resulting mask.

3.4 INTO [table] [AS options]

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 *options* argument is optional and can be a single value or a list, enclosed in square brackets, consisting of values and key=value. They can be used to specify the table and storage type. All keys and values are case-insensitive.

TYPE='value' specifies the table type.
PLAIN = make a persistent table, thus a true copy of all selected rows/columns.
SCRATCH = as plain, but only as a temporary table.
MEMORY = as plain, but keep everything in memory.
If *TYPE* is not given, a reference table is made if no expressions are given in the SELECT clause, otherwise a plain table is made.

ENDIAN='value' specifies the endianness
BIG = big endian
LITTLE = little endian
LOCAL = native endianness of the machine being used
AIPSRC = as defined in the .casarc file (which usually defaults to LOCAL)
If *ENDIAN* is not given, it defaults to AIPSRC.

STORAGE='value' specifies the storage type
SEPFILE = store as separate files (the old Casacore table format)
MULTIFILE = combine all storage manager files into a single file.
MULTIHDF5 = as MULTIFILE, but use an HDF5 file instead of a regular file.
DEFAULT = use SEPFILE (but might change in a future Casacore version),
AIPSRC = as defined in the .casarc file (which usually defaults to DEFAULT)
If *STORAGE* is not given, it defaults to AIPSRC.

BLOCKSIZE=n specifies the blocksize to use for MULTIFILE or MULTIHDF5.

OVERWRITE=F tells that an existing table with the given name should not be overwritten. By default TaQL will overwrite existing tables.

For backward compatibility, it is possible to specify an option directly without having to use 'key=value'.

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.

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.

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 if a result table cannot created. 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.

If the result is stored in a plain table, it is possible to give detailed data manager info for the result table using the DMINFO clause. See section 8.2 how the data manager info can be specified.

3.5 FROM table_list

The FROM part defines the tables used in the query. It is a comma-separated list of tables, each followed by an optional shorthand (alias).

The full syntax is:

```
FROM table1 [shorthand1], table2 [shorthand2], ...
```

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 precede the table name. It can be seen as an iterator variable.

The shorthand can be used in the query to qualify the table to be used for a column, for example `t0.DATA`. The first table in the list is the primary table which will be used if a column is not qualified by a shorthand. Often a query uses a single table in which case a shorthand is not needed. Multiple tables require a shorthand and are useful if:

- A keyword in another table is needed.
- Columns from multiple tables are used (an implicit join). In such a case the tables must have the same number of rows. For example, a regression test could be done like:

```
SELECT FROM test.MS t1, result.MS t2
  WHERE not all(near(t1.DATA, t2.DATA))
```

If the table is normal table with a fully alphanumeric name, the shorthand defaults to that name. In practice a shorthand is always needed if multiple tables are used.

The FROM clause can be omitted, in which case the input is a virtual table with no columns. The number of rows in it is defined by the LIMIT and OFFSET value; it defaults to 1 row. It makes it possible to select column-independent expressions in the SELECT command. Note that these expressions do not need to be constant. For example

```
SELECT rowid() LIMIT 31
```

creates a temporary table with column Col_1 and 31 rows containing the values 0..30.

A table can be given in a variety of ways.

1. A persistent table can be used by giving its name 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.

2. 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.

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

3. In a way similar to above, two colons can be used to denote the latest table at the same level. If none, at the next higher query level. The example below is an excerpt from the full example below.

   ```
   select from my.ms,
    [select from :: where sumsqr(UVW[1:2]) < 625] as TIMESEL
   ```

   The colons refer to the latest table used, thus `my.ms`.

4. 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 should be given using two colons instead of a slash like `my.ms::ANTENNA` which is a slight extension of specifying table names in the previous bullets.
   In this way a subtable can always be found.

5. Similar to 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 and CREATE TABLE command can also be used. The table created can thereafter be used in the rest of the query command by using the shorthand (alias) given to that table. It can also be used in the remainder of the table_list, thus using it as a backreference. Such backreferencing can be useful to avoid multiple equal subqueries. E.g.

```
select from my.ms,
 [select from :: 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 above 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.
Also note the use of :: in the second line which refers to `my.ms`.
Finally note that the new 3.2 is an easier way to use temporary tables.

6. 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.
   In a similar way as described above it is possible to use a subtable of such a table by specifying it as, for example, `$1::subtablename` or `$1.column::keyword`.

7. It is possible to use a concatenation of tables with the same description by giving a list of tables enclosed in square brackets. In this way it is, for example, possible to do a query on the combined parts of a MeasurementSet partitioned in time. Each table in the list can be specified in one of the ways mentioned in this section, including another table concatenation.
   For example:

   ```
   SELECT FROM [ms.part1, ms.part2, ms.part3] WHERE ...
   ```

   does a query on the three parts of an MS which are seen as a single table.
   It is possible to use glob filename patterns in such a list. For example

   ```
   SELECT FROM [ms.part*] WHERE ...
   ```

   is the same as the example above if no other files with such a name exist. An error is given if no table is found matching the pattern.

   Subtables of the concatenated tables can be concatenated as well. Alternatively, they can be assumed to be the same for all tables meaning that the subtable of the concatenation is the subtable of the first table. For example, when partitioning a MeasurementSet in time, the ANTENNA subtable is the same for all parts, while the POINTING and SYSCAL subtables depend on time, thus have to be concatenated as well. Concatenation of subtables can be achieved by giving them as a comma-separated list of names after the SUBTABLES keyword. For example:

```
SELECT FROM [ms.part1, ms.part2 SUBTABLES SYSCAL,POINTING]
```

Usually the result of a TaQL query references the table given in the FROM. In this example the FROM table is the concatenation, which is only known during the query. In such a case the concatenation must be made persistent, which can be done by using a GIVING (or INTO) inside the concatenation specification. Only the table name can be given, because the persistent concatenation only keeps the original table names; it does not make a copy of all data.
For example:

```
SELECT FROM [ms.part1, ms.part2 GIVING ms.conc]
   WHERE ANTENNA1 != ANTENAA2 GIVING ms.cross
```

selects the cross-correlation baselines from the concatenation. Note the two GIVING commands. The first one makes the concatenation persistent, the second one is the query result of the query *ms.cross*. It references the matching rows in the persistent concatenation *ms.conc* which in its turn references the original parts.

3.6 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.

3.7 GROUPBY group_list

It defines how rows have to be grouped. Usually a result per group will be calculated using aggregate functions. A group consists of all rows for which the columns (or expressions) given in the group_list have the same value. The (aggregate) expressions in the SELECT clause are calculated for the entire group. In this way one can get, for example, the mean XX amplitude and the number of time slots per baseline like:

```
SELECT ANTENNA1,ANTENNA2,GMEAN(AMPLITUDE(DATA[,0])),GCOUNT()
       FROM my.ms GROUPBY ANTENNA1,ANTENNA2
```

It results in a table containing *nbaseline* rows with in each row the antenna ids, mean amplitude, and number of rows.
If no aggregate function is used for a column, the value of the last row in the group is used. Note that in this example ANTENNA1 and ANTENNA2 are the same for the entire group. However, if TIME was also selected, only the last time would be part of the result.
Note that each expression in the group_list has to result in a scalar value of type bool, integer, double, date, or string.
Aggregate functions are discussed in more detail in section 5.

3.8 HAVING expression

This clause can be used to select specific groups. Only the groups (defined by GROUPBY) are selected for which the HAVING expression is true.
Note that HAVING can be given without GROUPBY, although that will hardly ever be useful. If no GROUPBY is given, but the SELECT statement contains an aggregate function, the result is a single group. HAVING cannot be used if neither GROUPBY nor SELECT aggregate functions are used.
It is discussed in more detail in section 5.

3.9 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. It operates on the output of the SELECT, thus after a possible GROUPBY and HAVING clause are executed.
The sort_list can be preceded 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
```

3.10 LIMIT/OFFSET expression

It indicates which of the matching and sorted rows should be selected. If not given, all of them are selected. The word `TOP` can also be used instead of `LIMIT`.
`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.

3.11 GIVING [table] [AS options] — 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 are 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. The next subsections discuss them in detail.

An expression can be used in several places:

- In the WHERE and HAVING clause where the result must be a boolean scalar value. It tells if a table row or group will be selected.
- As a key in the GROUPBY clause where the result must be a scalar value (numeric, bool, or string).
- 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 the 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.
- As a scalar or array value in various ALTER TABLE subcommands

The expression in the clause can be as complex as one likes using arithmetic, comparison, and logical operators. 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.
For example,

```
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/false (case-insensitive) or T/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 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   UC    UCHAR      BYTE
I2         SHORT      SMALLINT
U2   UI2   USHORT     USMALLINT
I4         INT        INTEGER
U4   UI4   UINT       UINTEGER
I8         INT8
R4   FLT   FLOAT
R8   DBL   DOUBLE
C4   FC    FCOMPLEX   COMPLEX
C8   DC    DCOMPLEX
```

```
S           STRING
TIME        DATE        EPOCH
```

The `TIME` type is a special data type. It means that the column gets data type `DOUBLE` and that a MEASINFO record will be defined in the column keywords to designate the column as an epoch.

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. 'E' or 'e' can be used to specify the 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 preceding them with a letter (e.g., mrad is millirad).
- A time/position in HMS format. Seconds can be left out. e.g., `12h34m34.5` or `8h32m`
- A position in DMS format. Seconds can be left out. e.g., `12d34m34.5` or `8d0m`
- A position as DMS in dot format. Note that all parts must be present. e.g., `12.34.34.5` or `8.0.34.5`

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 preceded by operator LIKE or NOT LIKE. Similar to PostgreSQL the operator ILIKE or NOT ILIKE can be used to indicate a case-insensitive comparison.

- A pattern or regular expression can be given like `x/expr/q` preceded 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:

  ```
  name ILIKE '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, possible conflicts can be solved by using whitespace in an 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, YYYY-MM-DD, or DD-MM-YYYY where DD and MM must be 2 digits and YYYY 4 digits.
   - DD-MMMMMMMM-YY where the - is optional and MMMMMMMM is the case-insensitive name of the month (at least 3 letters). DD can be 1 or 2 digits and YY 1 to 4 digits. 2000 is added if YY<50 and 1900 is added if 50<=YY<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.

   ```
   10-02-1997
   10-February-97
   ```

```
10feb97
1997/02/10          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.

```
[0:10]
['str1', 'str2', 'str3']
'str' + ['1', '2', '3']
```

The first example results in an integer array of 10 elements with values 0..9. The others result in a string array of 3 elements. The second version already shows that strings can be concatenated (as explained further on).

A multi-dimensional array can be formed by giving a set of arrays. A nested list resembles the *numpy* way. For example:

```
[[1,2,3],[4,5,6]]
```

results in a 2-dim array. However, it is also possible to use arrays created in other ways such as arrays in a column or arrays created with the `array` function described below. For example:

```
[[[1,2,3],[4,5,6]], array([10:13],2,3)]
```

results in a 3-dim array.

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, set, 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([1:11],10,4)
array([1:11], [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 reduce 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. Reduce functions like `median` only use the unmasked array elements. Furthermore, partial reduce 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. Both arrays must have the same shape. For example:

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

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

The functions `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.

The TaQL commands putting values into a table accept two columns (in parentheses) for a masked array. This is described in more detail in the appropriate sections. For example:

```
select means(DATA[FLAG],0) as (MD,MM) from in.ms giving out.tab
```

to write the data averaged over the first axis (frequency channel)into column MD. Only the unflagged data points are taken into account. The output contains the resulting flags in column MM; a flag is set to True if all channels were flagged.

4.3.10 Null Arrays

A cell in a table column containing variable shaped arrays, can be empty. Such a cell does not contain an array and is represented in TaQL as a null array. Note it is different from a cell containing an empty array, which is an array without values.

Null arrays can be used with any operator and in any function. If one of the operands or function arguments is a null array, the result will be a null array; only array functions reducing to a scalar (such as `sum` and `mean`) give a valid value (usually 0).

The UPDATE and INSERT commands will ignore a null array result; no value is written in that row.

4.4 Table Columns

A table column can be used in a query by giving its name in the expression, possibly qualified with a table shorthand name. 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 a column name. It can, however, be used 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.

4.4.1 Referring to SELECT columns

Usually a column used in an expression will be a column in one of the tables specified in the FROM clause. However, it is possible to use a column created in the SELECT clause, in expressions given in the HAVING or ORDERBY clause. In fact, a column name not preceded by a table shorthand, is first looked up in the SELECT columns and thereafter in the first FROM table.

It can be advantageous to use a SELECT column if that column is an expression; it saves both typing and execution time. because that expression is executed only once.

4.5 Table Keywords

It is possible to use table or column keywords, which can have a scalar or an array value or a record, possibly nested. 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, keywords in the primary table and in other tables can be used. If used from another 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 (I)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.
(I)LIKE	string	test if a string matches an SQL pattern (I for case-insensitive).
NOT (I)LIKE	string	test if string does not match an 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	`x BETWEEN b AND c` is similar to `x>=b AND x<=c` and `x 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 if the set contains more than one element.

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 the result 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 and intervals. 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 an integer, a real or a datetime. Incr has to be an integer or a real. Similar to Python an increment can be negative and start less than end. Some examples:

   ```
   1:10         means 1,2,...,9  (also 10 when using glish style)
   1:10:2       means 1,3,5,7,9
   10:1:-2      means 10,8,6,4,2
   1::2         means all odd numbers
   1:           means all positive integer numbers
   -1::-1       means all negative integer numbers
   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`, whitespace should be used as in `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.
   Note that an interval is not checked on `start<end`. If that is not the case, the result is an empty interval.

```
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])
```

If a set on the right side of the IN operator contains a single element (either a value, range, or interval), it does not need to be enclosed in square brackets or parentheses.

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. Similar to Python a negative value can be given meaning counting from the end. However, a negative stride cannot be given. Taking a single element can be done as:

```
array[1, 2]
array[-1, -1]                   last element
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)
[-1:-1,,]           last element of first axis, all elements other axes
```

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 real value which will be truncated to an integer.

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 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

TaQL has full support of units, both basic and compound units. Each value or subexpression can be followed by a unit telling that the value or subexpression result gets that unit or will be converted to that unit. All basic units supported by module Quanta can be used. Compound units (such as 'm/s') can be given as well or are formed by a TaQL expression with units (such as '10m/30s'). Note that units are case sensitive. Most common units use lowercase characters. A basic unit can be preceded by a scaling prefix (like `k` for kilo). The basic units and prefixes can be shown using the `show units` command of the program *taql*.

Most basic units can be given literally (i.e., as an unquoted string) after a value or subexpression. Whitespace between value or subexpression and unit is optional.
A compound 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 quoted (e.g., `'m/s'`) or escaped with a backslash (e.g., `m\/s`). Whitespace between value or subexpression and compound unit is mandatory unless the unit is quoted.
For example:

```
10.3deg      # 10.3 degrees
10.3 deg     # 10.3 degrees
(1+10)deg    # 11 degrees
3.6 'km/h'   # compound unit requires quotes (or backslash)
1/10 deg     # NOTE: results in unit deg-1 (divided by deg)!!!
(1/10)deg    # results in 0.1deg
```

Units can be converted to another (conforming) unit by giving that unit after a (sub)expression. E.g., `3 deg rad` converts 3 degrees to radians. Note that the empty string ('') is an empty unit, which can be used to make a value unitless. If ever needed, it can be used to set a non-conforming unit for a value. E.g. `(3deg '') kg`.

There is no real distinction between giving a unit as part of a value (as in `3deg`) or using whitespace between value and unit (as in `3 deg`). Also composite units (enclosed in quotes) can be given right after a value without whitespace.
However, a few units are identical to reserved TaQL keywords (e.g., 'in' for inch or 'as' for arcsecond). Such units have to be quoted or escaped with a backslash, unless given after a value without whitespace (as in `3in`). For example:

```
10deg rad         # is fine
```

```
10 deg rad        # is fine
10deg m           # error; non-conforming units
10deg '' m        # is fine (but makes little sense)
(10'm/s')'km/h'   # is fine
10'm/s' 'km/h'    # is fine
1in               # 1 inch
1 \in             # 1 inch
1 in              # error; in is a reserved keyword
1as deg           # 0.000277778 deg
(1 \as)deg        # 0.000277778 deg
(1 as)deg         # parse error; as is a reserved keyword
1 as deg          # very tricky; can result in column 'deg'
```

Arguments to functions such as `sin` are 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 to make their units the same. An exception is thrown if a unit conversion is not possible.

Units can be given (or derived) in various ways.

- A value can be followed by a unit.
- 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.
- If a column has a unit defined in column keyword `QuantumUnits` or `UNIT`, it automatically gets that unit.
- 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 such as `asin` give radians.
- When combining values with different units in e.g., an interval, a set, an addition, or a function such as `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. An error is given if the units do no conform.

  ```
  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
  3deg-4m          results in an error
  ```

- 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 for multiplication it is the unit of the argument with a unit and for division it is the unit of the first operand or the reciproke unit of the second operand.
  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`.

- Division of units of an equal kind (e.g., km by m) results in a unitless value.

- A value can be made unitless by using the empty string as a unit. This can be useful if values with different units have to be combined in a set.

```
1km/10m          result is 100
1/2s             result is 0.5 '(s)-1'
1Hz + 1/2s       result is 1.5Hz
```

- 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.

```
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 such as `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 200 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 values and relations 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 by making the first letter of each word uppercase and the rest of the word lowercase. A word is delimited by a non-alphanumeric character.

`string REVERSESTRING(string), string SREVERSE(string)`
: Reverse the characters of a string.

`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. N defaults to the string length. If the string argument is an array of strings, an array with the substring of each string is returned. The arguments ST and N have to be scalar values. If ST is negative, it counts from the end (a la Python). N and the resulting ST will be set to 0 if negative. If ST exceeeds the string length, an empty string is returned.

`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).
Note that the function `STR` discussed in the next section can also be used for pretty-printing a date/time. It gives more control over the number of decimals and date format.

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 STR(value, [format]), string STRING(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. Default precision is 6 (thus hh:mm:ss). The optional time/angle formats and modifiers are:

Format	Description
YMD	yyyy/mm/dd/hh:mm:ss.sss
YMD_ONLY	YMD without the time (same as YMD|NO_TIME)
DMY	dd-Mon-yyyy/hh:mm:ss.sss
FITS	yyyy-mm-ddThh:mm:ss.sss
BOOST	the same as DMY|USE_SPACE
NO_H, NO_D	suppress the output of hours (or degrees): useful for offsets
NO_HM, NO_DM	suppress the degrees and minutes
CLEAN	suppress leading or trailing periods or colons if not all time/angle parts are printed (e.g., when giving NO_H or 4 decimals)
DAY	precede the output with Day- (e.g., Wed-)
NO_TIME	suppress printing of time
ANGLE	+ddd.mm.ss.ttt
TIME	hh:mm:ss.ttt
USE_SPACE	use a space between date and time (and day and date)
DIG2	get angle/time in range -90:+90 or -12:+12
LOCAL	local time; in FITS mode append time zone as +hh:mm

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 datetime 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.
Note that operator `=` is the same as NEAR with a tolerance of 1e-5.

`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)`

`dnumeric ASIN(numeric)`

`dnumeric COS(numeric)`

`dnumeric COSH(numeric)`

`dnumeric ACOS(numeric)`

`dnumeric TAN(numeric)`

`dnumeric TANH(numeric)`

`dnumeric ATAN(numeric)`

`dnumeric ATAN2(numeric y, numeric 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 arguments or results of the trigonometric functions are in radians. They are converted automatically if units are given.

4.10.7 Array to scalar reduce 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.

`dnumeric MEAN(numeric), dnumeric AVG(numeric)`
Return mean of all elements.

`double VARIANCE(numeric)`
Return population variance (the sum of
`(a(i) - mean(a))**2/nelements(a)`.
Note that the variance of a complex value uses the absolute value and is the same as the sum of the variances of the real and imaginary parts.

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

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

`double SAMPLESTDDEV(numeric)`
Return sample standard deviation (the square root of the sample variance).

`double AVDEV(numeric)`
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, but no mean of the two middle elements is taken.

4.10.8 Array to array reduce 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,0,1)` 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,[1:10])` 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.

`dnumeric MEANS(numeric), dnumeric AVGS(numeric)`
: Return mean of elements.

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

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

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

`double SAMPLESTDDEVS(numeric)`
Return sample standard deviation (the square root of the sample variance).

`double AVDEVS(numeric)`
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 reduce functions described above, the axes must be given as the last argument(s) and 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?

`bool BOXEDNTRUE(bool)`
Return number of true elements.

`bool BOXEDNFALSE(bool)`
Return number of false elements.

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

`numeric BOXEDSUMSQUARE(numeric), numeric BOXEDSUMSQR(numeric)`
Return sum of squared elements.

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

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

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

`dnumeric BOXEDMEAN(numeric), dnumeric BOXEDAVG(numeric)`
Return mean of elements.

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

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

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

`double BOXEDSAMPLESTDDEV(numeric)`
Return sample standard deviation (the square root of the sample variance).

`double BOXEDAVDEV(numeric)`
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).

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

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 reduce functions described above, the axes must be given as the last argument(s) and 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?

`bool RUNNINGNTRUE(bool)`
Return number of true elements.

`bool RUNNINGNFALSE(bool)`
Return number of false elements.

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

`numeric RUNNINGSUMSQUARES(numeric), numeric RUNNINGSUMSQR(numeric)`
Return sum of squared elements.

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

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

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

`dnumeric RUNNINGMEAN(numeric), dnumeric RUNNINGAVG(numeric)`
Return mean of elements.

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

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

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

`double RUNNINGSAMPLESTDDEV(numeric)`
Return sample standard deviation (the square root of the sample variance).

`double RUNNINGAVDEV(numeric)`
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).

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

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 represents 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 represents 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 represents 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

The following functions create an array value with or without a mask. Function `marray` creates a new masked array, the other functions return a masked array if the input was masked, otherwise an unmasked array.

`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 MARRAY(anytypearray,boolarray)`
: This function offers another way to create a masked array. The mask must be given in the second argument; its shape must be the same as the shape of the data array. If the argument is a scalar, it returns a 1-dim array with one element.

`anytypearray NULLARRAY(anytype)`

This function creates a null array. Its data type is determined by the data type of the argument. The argument value itself is not used. It is mainly meant for test purposes.

`anytypearray RESIZE(anytypearray,newshape[,mode])`

This function resizes an array to the given shape and copies the values. The optional `mode` argument determines how the values are copied. If the argument is not given, the new shape is arbitrary and the dimensionality can change. The values are copied to the same index in the new array. If an axis gets larger, the new values are set to 0 (or an empty string).

If mode is given, the dimensionality can change as well, but each new axis has to be a multiple of the old one. If the dimensionality grows, the missing input axes have length 1. If `mode=0`, copying the values is done in an upsampling way. E.g., if a new axis is twice the length of the old one, values 1,2,3 are copied as 1,1,2,2,3,3. A good use case is applying the flags of averaged data to the original data. If `mode=1`, the values in the example above are copied repeatedly as 1,2,3,1,2,3. By giving the mode as a set, it is possible to specify the mode per axis, but that is quite esoteric.

`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 REVERSEARRAY(anytypearray[,axes])` (or `AREVERSE`)

This function reverses the elements of the given axes in an N-dim array. For example, reversing the outer axis of [[0,1],[2,3]] results in [[2,3],[0,1]], while reversing the inner axis results in [[1,0],[3,2]] If no axes are given, all axes are done resulting in a fully reversed array ([3,2],[1,0]] in the example).

A possible mask is reversed 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 negative value means below the main diagonal, while positive means above the main diagonal.

A possible mask is diagonaled as well.

`anytypearray ARRAYDATA(anytype)`

This function returns the array without a mask, thus removes the mask. If the argument is a scalar, it returns a 1-dim array with one element.

`boolarray ARRAYMASK(anytype), boolarray MASK(anytype)`

This function returns the mask of an array. If the array has no mask, it returns a boolean array of the same shape with all values set to False. If the argument is a scalar, it returns an 1-dim array with one False element.

`anytypearray NEGATEMASK(anytype)`

This function returns the array with the negated mask. If the array has no mask, it returns the

array with a mask of all Trues. If the argument is a scalar, it returns a 1-dim array with one element.

`anytypearray REPLACEMASKED(anytype, anytype)`
This function replaces the masked elements in the first argument by the corresponding value in the second argument (which can be a scalar value). If the first argument has no mask, the function is a no-op. If the first argument is a scalar, it returns a 1-dim array with one element.

`anytypearray REPLACEUNMASKED(anytype, anytype)`
This function replaces the unmasked elements in the first argument by the corresponding value in the second argument (which can be a scalar value). If the first argument has no mask, the replacement value is returned; a scalar replacement is expanded to the array shape. If the first argument is a scalar, it returns a 1-dim array with one element.

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

4.10.13 Aggregate functions

The `GXXX` aggregate functions calculate an aggregated value for all rows in a group, usually defined with a GROUPBY clause. For example, when grouping in TIME, an aggregate function like `GNTRUE(FLAG)` counts per time slot the number of flagged data points. Aggregate functions can only be used in the SELECT and the HAVING clause.
Most functions listed below reduce the values in a group to a scalar value, also if the value in a row is an array (as in the GNTRUE example above). The arrays in a group can have different shapes.

However, there are several aggregate functions returning an array as done by the last three functions (GHIST, GAGGR, and GROWID) shown below. Furthermore, most scalar functions have a plural form (e.g., GNTRUES) returning an array. They are described at the end of this section.

Note that the aggregate function names differ from their SQL counterparts; they all have the prefix `G`, because TaQL functions like `MAX` already exist for array operations. This naming scheme also makes it more clear which TaQL functions are aggregate functions.

A technical detail is how aggregate functions are implemented. TaQL walks sequentially through a table. Non-lazy functions operate directly on the value in a row making the table access purely sequential. It requires that the results of all groups are held in memory. For some functions, in particular GAGGR, this could lead to a very high memory usage. Therefore, some functions are implemented in a lazy way. They keep the row numbers of a group and access the data when the aggregated result of a group is needed. In this way only the data of a single group needs to be held in memory, but the access to the table might be non-sequential making it somewhat slower. Currently, only GAGGR and the User Defined aggregate functions are implemented in a lazy way.

`integer GCOUNT(), integer GCOUNT(*)`
Return the number of rows per group.

`integer GCOUNT(columnname)`
Return the number of rows per group for which the column has a value. Note that only a column containing variable sized arrays can contain empty cells.

`anytype GFIRST(anytype)`
Return the first value of an expression in the group. The values of a column not mentioned in

the GROUPBY clause, might differ. This function can be used to return the value of the first row in the group.

`anytype GLAST(anytype)`
Return the last value of the group (is similar to GFIRST).
Note this function is implicitly used if an expression without aggregate function is used in a group.

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

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

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

`integer GNFALSE(bool)`
Return number of false elements.

`numeric GSUM(numeric)`
Return sum of all elements.

`numeric GSUMSQUARE(numeric), numeric GSUMSQR(numeric)`
Return sum of all squared elements.

`numeric GPRODUCT(numeric)`
Return product of all elements.

`real GMIN(real)`
Return minimum of all elements.

`real GMAX(real)`
Return maximum of all elements.

`dnumeric GMEAN(numeric), dnumeric GAVG(numeric)`
Return mean of all elements.

`double GVARIANCE(numeric)`
Return population variance (the sum of
`(a(i) - mean(a))**2/nelements(a)`.

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

`double GSTDDEV(numeric)`
Return population standard deviation (the square root of the variance).

`double GSAMPLESTDDEV(numeric)`
Return sample standard deviation (the square root of the sample variance).

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

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

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

`double GHIST(real, intscalar nbin, realscalar start, realscalar end)`
: Return the histogram of the data using the given number of bins. The histogram contains an extra bin at the beginning and the end for the outliers. If the rows in the group contain arrays, they can have variable shapes.

`anytypearray GAGGR(anytype), anytypearray GSTACK(anytype)`
: Stack the row values in a group to form an array where the row is the slowest varying axis (similar to numpy's dstack). Thus if the column contains scalar values, the result is a vector. Otherwise it is an array whose dimensionality is one higher. It requires that all arrays in a group have the same shape.
  Note that this function can be very useful for arrays, because it makes it possible to use partial reduce functions like `medians` to calculate the medians along arbitrary axes.

`integerarray GROWID()`
: Return the row numbers of the rows in the group.

Most functions above have a plural counterpart. They calculate the aggregated value per array index, thus the result has the same shape as the arrays in the group. Similar to function GAGGR, they require that all arrays in a group have the same shape.
For instance, for a MeasurementSet the expression `GMEANS(DATA)` calculates the mean in a group per channel/polarization. Not only it is a shorthand for `MEANS(GAGGR(DATA), 0)`, but it usually works faster because, unlike GAGGR, it is non-lazy.
The functions available are:

```
GANYS   GALLS       GNTRUES       GNFALSES
GMINS   GMAXS
GSUMS   GPRODUCTS   GSUMSQRS      GSUMSQUARES
GMEANS  GAVGS       GVARIANCES    GSTDDEVS
GRMSS           GSAMPLEVARIANCES  GSAMPLESTDDEVS
```

4.10.14 Miscellaneous functions

`bool ISNULL(anytype)`
: Return True if the argument value is a null array (an array with 0 axes). Note that function `NULLARRAY` can create a null array.

`bool ISDEFINED(anytype)`
: Return False if the array value in the current row is undefined (is null). It makes it possible to test if a cell in a column with variable shaped arrays contains an array. Furthermore, it can be used to test if a field in a record is defined.
  Note that function `ISNULL` can also be used to test for an undefined array in a row.

`bool sh.ISCOLUMN(string)`
Return False if no column with the given name exists in the table with the shorthand given before the function name. If no shorthand is given, the first table will be used.

`bool sh.ISKEYWORD(string)`
Return False if no keyword with the given name exists in the table with the shorthand given before the function name. If no shorthand is given, the first table will be used. The keyword name can be given as described in section 4.5, thus the name of a table keyword or column keyword or a nested field can be specified.

`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 4.11 and examples in section 12.1).

`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.

4.10.15 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 python style, element `(i,j)` in the resulting array is `T` if source `i` is contained in cone `j`.

`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.

4.10.16 User defined functions

TaQL can be extended with so-called User Defined Functions (UDF). These are dynamically loaded functions, either written in C++ or in Python. In TaQL the name of a UDF written in C++ consists of the name of the library (without lib prefix and extension) followed by a dot and the function name. For example:

```
meas.hadec(...)
```

denotes function `hadec` in shared library `libmeas.so` or `libcasa_meas.so`. For OS-X the extension `.dylib` will be used.
The physical shared library name must be fully lowercase, but the UDF name used in TaQL is case-insensitive. The name of a UDF written in Python is like `py.module.func` where the module part is optional. In the `USING STYLE` clause it is possible to define synonyms for the UDF library names. By default, `mscal` is defined as a synonym for `derivedmscal` and `py` as a synonym for `pytaql`.

Usually a UDF will operate on the arguments given to the function and will not itself operate on a table given in a query command. However, some UDFs (most notably the `mscal` ones) do not have arguments, but operate directly in a specific way on a table. Normally they use the first table given in the FROM clause, but the UDF name can be preceded by a table shorthand to specify another table. For example:

```
select t1.mscal.ha1(), t2.mscal.ha1() from my1.ms t1, my2.ms t2
```

to get the hourangle from two different tables. Of course, both tables need to have the same number of rows.
Note that UDFs not directly operating on a table, will ignore a shorthand.

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

4.10.17 Special MeasurementSet functions

The Casacore package comes with several predefined UDFs in library `libcasa_derivedmscal`. It contains four groups of UDFs, all operating on a MeasurementSet and several on a CalTable, the CASA calibration table (both old and new format).
Although the library is called `derivedmscal`, for ease of use it is possible to use the synonym `mscal`.

Get derived values

The first group calculates derived values like hourangle and azimuth for each row in the MeasurementSet or CalTable given in the FROM clause. It uses the time, direction and arraycenter or first or second antenna of a baseline from the MeasurementSet or CalTable. For a CalTable, where a row contains a single antenna, functions like PA1 are the same as PA2. All angles are returned in radians.

`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 topocentric hourangle/declination of the array center (observatory position).

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

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

`doublearray MSCAL.AZEL()`
gives the topocentric azimuth/elevation of the array center (observatory position).

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

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

`doublearray MSCAL.ITRF()`
gives the direction in (time-dependent) ITRF coordinates.

`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.NEWUVW()`
gives the 3-vector of UVW coordinates in J2000 in meters. It recalculates them, thus does not return the UVW coordinates stored in the MeasurementSet.

`doublearray MSCAL.NEWUVWWVL()`
gives the 3-vector of calculated UVW coordinates in J2000 in wavelengths for the reference frequency of the appropriate spectral window.

`doublearray MSCAL.NEWUVWWVLS()`
gives the nfreq*3-matrix of calculated UVW coordinates in J2000 in wavelengths for all channel frequencies of the appropriate spectral window.

`doublearray MSCAL.UVWWVL()`
gives the 3-vector of stored UVW coordinates in wavelengths for the reference frequency of the appropriate spectral window.

`doublearray MSCAL.UVWWVLS()`
gives the nfreq*3-matrix of stored UVW coordinates in wavelengths for all channel frequencies of the appropriate spectral window.

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 in the FIELD table by giving its name as a string argument (e.g., HA('DELAY_DIR')).
Except for the last 2 functions, it is 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) or a known source (such as CygA). For example:

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

The examples above give the azimuth and elevation of the given directions for each selected row in the MeasurementSet, 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.

Stokes conversion

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.

CASA style selection

The `BASELINE` function makes it possible to do selection on baselines in a MeasurementSet or CalTable using the special CASA selection syntax described in note 263. Similar functions `CORR, TIME, FIELD, FEED, SCAN, SPW, UVDIST, STATE, OBS`, and `ARRAY` can be used to do selection based on other meta data. The functions accept a string containing a selection string and return a Bool value telling if a row matches the selection string. For example,

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

selects the cross-correlation baselines containing an antenna whose name matches the pattern in the function argument.
Note there is a difference how CASA and TaQL handle unknown antennas given in the baseline selection string. CASA tasks give an error, while TaQL will not complain and not even report it, because doing a selection this way should not behave differently from doing it like `NAME='RTX'`.
Also note that in CASA tasks only one selection string per type can be given and the final selection is the AND of them. TaQL has the AND and OR operators making it possible to combine the selections in all kind of ways, possibly using multiple selection strings of the same type.

Get values from a subtable

Several functions exist to get information like the name of an antenna from the subtable for each row in the main table. Basically they do a join of the main table and a subtable. For example:

```
select mscal.ant1name(), mscal.ant2name() from my.ms
```

gets the names of the antennae used in each baseline.

The following functions can be used:

`string MSCAL.ANT1NAME()`
gives the name of ANTENNA1.

`string MSCAL.ANT2NAME()`
gives the name of ANTENNA2.

`anytype MSCAL.ANT1COL(ColumnName)`
gives for ANTENNA1 the value in the given column (in quotes) in the ANTENNA subtable.

`anytype MSCAL.ANT2COL(ColumnName)`
gives for ANTENNA2 the value in the given column (in quotes) in the ANTENNA subtable.

`anytype MSCAL.STATECOL(ColumnName)`
gives for STATE_ID the value in the given column (in quotes) in the STATE subtable.

`anytype MSCAL.OBSCOL(ColumnName)`
gives for OBSERVATION_ID the value in the given column (in quotes) in the OBSERVATION subtable.

`anytype MSCAL.SPWCOL(ColumnName)`
gives for DATA_DESC_ID the value in the given column (in quotes) in the SPECTRAL_WINDOW subtable.

`anytype MSCAL.POLCOL(ColumnName)`
gives for DATA_DESC_ID the value in the given column (in quotes) in the POLARIZATION subtable.

`anytype MSCAL.FIELDCOL(ColumnName)`
gives for FIELD_ID the value in the given column (in quotes) in the FIELD subtable.

`anytype MSCAL.PROCCOL(ColumnName)`
gives for PROCESSOR_ID the value in the given column (in quotes) in the PROCESSOR subtable.

`anytype MSCAL.SUBCOL(SubtableName, ColumnName, idcolumn)`
gives for the (integer) id-column the value in the given column in the given subtable. This is the most common form and can be used to join any table with a subtable.

Note that the following are equivalent. The first versions are shorthands for the latter ones.

```
mscal.ant1name()
mscal.ant1col('NAME')
mscal.subcol('ANTENNA', 'NAME', ANTENNA1)
```

In the last example the id-column must be given as such, thus must not be a string.

4.10.18 Special Measures functions

These functions make it possible to convert Casacore measures (e.g., directions) from one reference frame to another. The prefix `MEAS.` has to be used for all these functions. The MEAS library libcasa_meas.so (or .dylib) will be loaded if not loaded yet. All conversions supported by Casacore's Measures are possible. It is quite flexible; for instance, source names can be used instead of right ascension and declination. Also it recognizes nested MEAS functions and table columns containing measures. For example:

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

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

Below it is described how the measure values can be specified. Further down it is described in detail for each measure type.

- Measures must be given as values optionally followed by a case-insensitive string defining the reference frame of the values. If no frame is given, a default frame is assumed. Some value specifications imply a measure frame, in which case the frame should not be specified. If specified, it should match the implied frame.
  The possible frames depend on the measure type. They can be shown using the `show meastypes` command in the program *taql*.

- Values can be given as expressions with constant values and/or table columns. The values can have a unit and sometimes must have a unit. If no unit is given, it is implied (similar to the reference frame) or a default unit is assumed.

- A constant value can be a numeric expression, but it can also be a constant string defining the name of an observatory (for a position), a celestial source (for a direction), or a line (for a frequency) which are defined in the Measures tables. In such a case the measure frame and unit are implicitly defined and should not be given.
- Depending on the measure type, a measure is represented by a single value or multiple values (e.g., 2 or 3 for positions and directions).
- If the value is a table column containing measures, the measure frame is obtained from the column and no extra frame argument should be given.
- If the value is a nested MEAS function, it recognises its type and measure frame and no extra frame argument should be given.

Many functions are available, but they come down to a few basic functions. The others (described further down) are synonyms or shorthands for the basic functions described below.
A function will operate on each element of the Cartesian product of the function arguments.

`doublearray MEAS.POSITION(toref, position)`
converts positions to the reference frame given by the 'toref' string. The `toref` value can have a suffix telling how to return the result.
- None or XYZ: return as xyz (unit m).
- LLH: return as lon-lat-height (in rad,rad,m but without units).
- LL or LONLAT: return as lon-lat (unit rad).
- H or HEIGHT: return as height (unit m).

`doublearray MEAS.EPOCH(toref, epoch, position)`
converts epochs to the reference frame given by the 'toref' string and returns the values with unit 'd'. The position argument only needs to be given if the conversion depends om position (e.g., UTC to LST). By default conversions to sidereal time (e.g., LAST) return the fraction giving the true sidereal time. Only if the `toref` string starts with 'F-', 'F_', 'f-', or 'f_' the full sidereal time is returned which includes the number of sidereal days since the start of MJD.

`doublearray MEAS.DIRECTION(toref, direction, epoch, position)`
converts directions to the reference frame given by the 'toref' string and returns them as angles with unit 'rad'. The epoch and position arguments only need to be given if the conversion needs such information (e.g., when converting J2000 to apparent).

`doublearray MEAS.DIRCOS(toref, direction, epoch, position)`
Same as function DIRECTION, but returning 3 direction cosines instead of 2 angles.

`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 epochs and positions.
Note that the source can be invisible all day (results in set<rise). If visible all day, rise time is 0h0m and set time is 24 hours later.
The TIME or CTIME function can be used on the result to get the time part only (as double cq. string).
If the Sun is used as a source name (case-insensitive), it can be followed by a hyphen and one

of the following case-insensitive suffices indicating which part of the sun to use. The default is UR which is used in most almanacs.

- `CR`: use the center of the sun with refraction correction.
- `UR`: use the upper brim of the sun with refraction correction, thus show when part of the sun is visible.
- `LR`: use the lower brim of the sun with refraction correction, thus show when the full sun is visible.
- `C`: use the center of the sun without refraction correction.
- `U`: use the upper brim of the sun without refraction correction.
- `L`: use the lower brim of the sun without refraction correction.
- `CT`: use the civil twilight (6 deg).
- `NT`: use the nautical twilight (12 deg).
- `AT`: use the amateur astronomical twilight (15 deg).
- `ST`: use the scientific astronomical twilight (18 deg).

The first six suffices can also be used with the Moon.
See stjarnhimlen.se for additional information.

`doublearray MEAS.IGRF(toref, height, direction, epoch, position)`
calculates earth magnetic field values using the IGRF model and returns them in the reference frame given by 'toref'. A suffix to the function name determines how the values are returned.
- None or XYZ: return as xyz (unit nT).
- LL or LONLAT: return as lon-lat (unit rad).
- STR or STRENGTH: return as strength (unit nT).

`doublearray MEAS.IGRFLOS(height, direction, epoch, position)`
calculates earth magnetic field values using the IGRF model along the line of sight. The result is in unit nT.

`doublearray MEAS.IGRFLONG(height, direction, epoch, position)`
calculates the longitude of the IGRF model calculation. The result is in radians.

`doublearray MEAS.EARTHMAGNETIC(toref, em, epoch, position)`
converts earth magnetic field values to the reference frame given by the 'toref' string. Similar to function IGRF the function name suffix (XYZ, LL or STR) determines how the result is returned.

`doublearray MEAS.FREQUENCY(toref, freq, radvel, direction, epoch, position)`
converts frequencies to the reference frame given by the 'toref' string and returns them with unit Hz. Parameter 'radvel' is only needed when converting to/from rest frequencies. The frequency can be specified as period or wavelength by using the proper units. If no unit is given, Hz is assumed.

`doublearray MEAS.RESTFREQUENCY(freq, doppler)`
calculates the rest frequencies from the doppler shifts and returns them with unit Hz.

`doublearray MEAS.SHIFTFREQUENCY(freq, doppler)`
shifts the (rest) frequency according to the doppler shift and returns them with unit Hz. (opposite of function RESTFREQUENCY).

`doublearray MEAS.DOPPLER(toref, doppler)`
converts the dopplers to the 'toref' type.

`doublearray MEAS.DOPPLER(toref, radvel)`
calculates the dopplers from the radial velocities.

`doublearray MEAS.DOPPLER(toref, freq, restfreq)`
calculates the dopplers from the frequencies and rest frequencies.

`doublearray MEAS.RADIALVELOCITY(toref, radvel, direction, epoch, position`
converts the radial velocities to the 'toref' frame and return them with unit km/s.

`doublearray MEAS.RADIALVELOCITY(toref, doppler)`
calculates the radial velocities from the dopplers and returns them with unit km/s.

For ease of use several functions have a shorthand synonym.

- POS for POSITION
- DIR for DIRECTION
- EM for EARTHMAGNETIC
- FREQ for FREQUENCY
- REST or RESTFREQ for RESTFREQUENCY
- SHIFT or SHIFTFREQ for SHIFTFREQUENCY
- REDSHIFT for DOPPLER
- RV or RADVEL for RADIALVELOCITY

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

`doublearray MEAS.ITRFxxx(position)`
converts a position to ITRF coordinates, where xxx must be XYZ, LLH, LONLAT, LL, HEIGHT or H to define the result.

`doublearray MEAS.WGSxxx(position)`
converts a position to WGS84 coordinates, where xxx is the same as above. If xxx is omitted, it defaults to XYZ.

`doublearray MEAS.LAST(epoch, position)`
converts an epoch to local sidereal time (without date part).
Function name MEAS.LST can be used as well.

`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.ITRFD(direction, epoch, position)`
converts a direction to ITRF coordinates.
Function name MEAS.ITRFDIR or MEAS.ITRFDIRECTION can be used as well.

`doublearray MEAS.LSRK(frequency, direction, epoch, position)`
converts a frequency to the LSRK frame.

`doublearray MEAS.LSRD(frequency, direction, epoch, position)`
converts a frequency to the LSRD frame.

`doublearray MEAS.BARY(frequency, direction, epoch, position)`
converts a frequency to the BARY frame.

`doublearray MEAS.REST(frequency, radvel, direction, epoch, position)`
calculates the rest frequency.
Function name MEAS.RESTFREQ or MEAS.RESTFREQUENCY can be used as well.

The function arguments can be given in a variety of ways. Coordinate values (such as directions) can be followed by a 'ref' argument telling the reference frame used for them (e.g., J2000). If not given, a default reference frame is assumed.
Where needed, the argument data types and units are used to distinguish arguments. However, a string value for a reference frame cannot be distinguished from a string giving the name of a source, observatory or line. In such a case the string value is used to distinguish them.

- 'toref' is a constant scalar string giving the reference frame to convert to. The command

  ```
  taql show meastype
  ```

  can be used to see the possible frames for each Measure type.

- 'direction' gives one or more directions. They can be given in several ways.

- An array of directions, each 2 angles or 3 directions cosines. It can be given as a single list or a multi-dim array. The choice between angles and direction cosines is based on the size of the first dimension. If divisible by 2, it is angles, by 3 is direction cosines, otherwise an error. Thus a list of 6 elements defines 3 directions with 2 angles each (default in radians). It can be followed by a string defining the source reference frame which defaults to 'J2000'.
- If a single constant direction is used, it can be given as 2 (for angles) or 3 (for direction cosines) scalar values, followed by the optional source reference frame.
- The name of a column in a table or a subset of it such as `DELAY_DIR[0,]`. Often this is a TableMeasures column which is recognized as such, also its source reference frame. If such a column is given as part of an expression, it will not be recognized as a TableMeasures column and its reference frame should be given.
- As a list of (case-insensitive) names of planetary objects (such as SUN or JUPITER) or names of standard sources. (CasA, CygA, TauA, VirA, HerA, HydA, or PerA). ZENITH can be given as well. In the future support for comets might be added.

For example:

```
['MOON','sun', 'venus']          # 3 planetary objects
12h23m17.5, 23d56m43.8, 'B1950'  # ra/dec as scalar constants (as B1950)
[12h23m17.5, 23d56m43.8]         # ra/dec as array (default J2000)
PHASE_DIR[0,]                    # direction ra/dec 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 expression 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
9Sep2011/12:00:00, 'UTC'            # given UTC time
```

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

- 'position' gives one or more earth positions 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 distinguish between LL and XYZ by using suffices such as XYZ in the reference frame type given in the next argument.
  - An array of positions given as xyz, as lonlat, as lon-lat-height or as height. The latter is taken towards the pole. Note that specifying as lon-lat-height precludes use of units (angle and length units cannot be mixed in a TaQL value). It can be given as a single list or as

a multi-dim array. If given as lonlat it can be followed by an array defining the height for each lon,lat pair (their sizes should match). Finally it can be followed by a string defining the source reference frame type, which defaults to ITRF for xyz and WGS84 for lonlat. The source reference frame type can contain the suffix XYZ, LLH, LL or H to tell how the values are specified. If no suffix is given, it is derived from the unit of the first value (angle means LL, length means XYZ).

- If a single constant position is used, it can be given as 1, 2 or 3 scalar values, optionally followed by the source reference type. If xyz, lonlat, lon-lat-height or height is given is derived in the same way as above.
- The name of a column in a table or a subset of it such as `POSITION[0,]`. Often this is a TableMeasures column which is recognized as such, also its source reference frame. If such a column is used in a expression, it will not be recognized as a TableMeasures column and its reference frame should be given.
- A list containing (case-insensitive) names of known observatories such as 'WSRT' or 'VLA'.

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

```
'WSRT'                              # WSRT position
5deg, 52deg                         # 2 scalar constants (WGS84 lonlat)
(5deg, 52deg]                       # same, but as array
5deg, 52deg, 5m                     # WGS84 lonlat 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
```

- 'em' gives the earthmagnetic values to use, thus earth magnetic field strengths for given points. They can be given as xyz or as lon-lat-strength. The default unit is nT.

  - An array of earthmagnetics given as xyz or as lon-lat-strength Note that specifying as lon-lat-strength (lls) precludes use of units (angle and length units cannot be mixed in a TaQL value), while xyz must have a proper unit (e.g., nT or G). It means that the unit determines if xyz or lon-lat-strength is given. It can be given as a single list or a multi-dim array. It can be followed by a string defining the source reference type, which defaults to ITRF.
  - If a single constant earthmagnetic is used, it can be given as 3 scalar values, optionally followed by the source reference type. The unit defines if xyz or lon-lat-strength is given.
  - The name of a column in a table or a subset of it such as ¡src¿EMVAL[0,]¡/src¿. Often this is a TableMeasures column which is recognized as such, also its source reference frame. If such a column is used in a expression, it will not be recognized as a TableMeasures column and its reference frame should be given.

  For example:

```
43deg, 45deg, 10nT, 'J2000'         # as scalars in J2000
[43, 45, 10], 'J2000'               # as array in J2000
```

- 'frequency' gives one or more frequencies 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 LSRK.
  Frequencies can be given as a scalar or array containing double values. The type of wave characteristics is recognized from the unit (using Casacore's MVFrequency class). The values are converted to proper frequencies (in Hz).
  - frequency (1/time)
  - time
  - angle/time (in 2pi units)
  - wavelength
  - 1/wavelength (in 2pi units)
  - energy (h.nu) (mass*length*length/time/time using Planck's constant)
  - impulse (mass*length using Planck's constant and speed of light)

  For example:

```
1e7                             # 10 MHz in LSRK
10MHz, 'BARY'                   # 10 MHz in BARY
10m                             # 29.9792 MHz in LSRK
10s, 'LSRK'                     # 0.1 Hz
```

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

- 'radialvelocity' gives one or more radial velocities. Its default unit is km/s and its default reference frame is LSRK. Similar to the other values, they can be given as a scalar, as an array or as a column (expression).

  For example:

```
[200,300]'km/s', 'BARY'         # array of 2 velocities in BARY
```

- 'doppler' gives one or more (unitless) doppler shifts. Its default reference type is RADIO. Similar to the other values, they can be given as a scalar, as an array or as a column (expression). For example:

```
2.5, 'RADIO'                    # RADIO doppler shift
```

Below a few examples are given showing how the MEAS functionality can be used.

```
meas.lst (15Oct2011/15:34, 5deg, 52deg)
hms(meas.lst ('15Oct2011/15:34', 'WSRT'))
```

calculates the local apparent sidereal time for the given date/time and position. The second example shows that an observatory name can be used for the position. It also shows that the date/time can be given as a string.

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

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'
```

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

```
meas.azel([03h13m10,65d50m12], 24sep2015/12:0:0+[0:24]h, 'LOFAR') deg
dms(meas.azel(03h13m10,65d50m12,'B1950', 24sep2015/12:0:0+[0:24]h, 'LOFAR'))
```

calculates the azimuth/elevation of the given source direction for the LOFAR site for the next 24 hours on the given date. The result is an array with shape [24,2]. The direction in the second example is given in B1950, the first as the default J2000. The result of the first example is double values with unit deg (given at the end of the expression). The result of the second example is strings in DMS format (because function DMS is used).

```
meas.rest(1.03GHz,'LSRK', 210'km/s','LSRK', [03h13m10,65d50m12],'J2000',
          2Jul2016/13:3:41, 'WSRT')
```

calculates the rest frequency for the given radial velocity, direction, date/time and position. The result will have unit Hz.

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 and 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 mscal.ant1name() from ~/GER1.MS
```

is a newer and easier way to obtain the name of ANTENNA1. It makes use of the new user defined functions in derivedmscal which can do an implicit join of a MeasurementSet and its subtables.

```
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 MeasurementSets.

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 an 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 3.11). 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.
   Using the new derivedmscal functions, below a much nicer solution is given.

6. `select from MY.MS where any(mscal.uvwwvls() < 100)`

   It shows how the UVWWVLS function in `derivedmscal` can be used to obtain the UVW coordinates in wavelengths.

7. Calculate the angular distance between the Mars and Jupiter as seen from the WSRT for the coming 30 days.

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

5 Aggregation, GROUPBY, HAVING

Similar to SQL it is possible to do aggregation and grouping in TaQL and to do selection on the groups using the HAVING clause.

5.1 Aggregation and GROUPBY

One or more aggregated values can be calculated for a group defined by the GROUPBY clause. The aggregate functions described in section 4.10.13 can be used. For example:

```
SELECT ANTENNA1, ANTENNA2, gcount(), sqrt(sumsqr(UVW[:2]))
    FROM my.ms GROUPBY ANTENNA1,ANTENNA2
```

A group is formed for the unique values of the columns given in the GROUPBY clause. In the example above a group per baseline is formed. Usually an aggregate function is ued to calculate a value for the group. In the example above the aggregate function `gcount()` counts the number of rows per baseline.
Often only the GROUPBY columns and aggregated values are part of the SELECT clause, but the example shows that other values (here the baseline length) can also be selected. Non-aggregated values get the values in the last row of a group.

Usually aggregated values and GROUPBY are used jointly, but it is possible to leave out one of them. If GROUPBY is not given, the entire table is a single group. For example:

```
SELECT gcount() from my.ms
```

does not have groups, thus shows the total number of rows in the MS.

```
SELECT ANTENNA1,ANTENNA2 from my.ms GROUPBY ANTENNA1,ANTENNA2
```

does not use aggregate functions, but shows the unique baselines in the MS. Apart from the order, it has the same result as

```
SELECT ANTENNA1,ANTENNA2 from my.ms ORDERBY UNIQUE ANTENNA1,ANTENNA2
```

but is somewhat faster.

In the examples above a sole aggregate function is used, but it is also possible to use it in an expression. Similarly, an expression can be used in the GROUPBY. For example:

```
select ctod(gmean(TIME)), gcount() from ~/data/GER.MS
  groupby round((TIME -
    [select gmin(TIME) from ~/data/GER.MS][0])/INTERVAL/5)
```

groups the MS in chunks of 5 time slots. Note that the nested query gets the TIME of the first time slot. The result is a set, hence the 0th element has to be taken.

Note that an aggregate function can only be used in the SELECT and HAVING clause, so TaQL will give an error message if used elsewhere.

5.2 HAVING

The HAVING clause can be used to select specific groups. For example:

```
SELECT TIME, gmax(amplitude(DATA)) as MAXA from my.ms GROUPBY TIME
    HAVING MAXA > 100
SELECT TIME, gmax(amplitude(DATA)) from my.ms GROUPBY TIME
    HAVING gmax(amplitude(DATA)) > 100
```

groups by time, but only selects the groups for which the maximum amplitude of the DATA is more than 100. Both examples give the same result, but the first one is more efficient. Not only it is less typing, but it is faster because it reuses the result column MAXA of the SELECT part.
Similar to WHERE, any expression can be used in HAVING, but the result has to be a bool scalar value.

As shown in the example, HAVING will normally use aggregate functions, but it is not strictly needed. However, selections without an aggregate function could as well be done in the WHERE clause.
Usually HAVING will be used in combination with GROUPBY, but it can be used without. It can also be used without an aggregate function in the SELECT. However, it is an error if both are omitted.

6 Some further remarks

6.1 Joining tables

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

6.1.1 Join on row number

```
SELECT FROM mytable t1,othertable t2
  WHERE not all(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 in corresponding rows are compared.

6.1.2 Join using an indexed subquery

```
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.

6.1.3 Join using a subquery set

```
SELECT FROM ~/GER1.MS WHERE ANTENNA1 IN
       [SELECT ROWID() FROM ::ANTENNA WHERE NAME ~ p/CS*/]
```

This example shows another way to use a subquery for a join of the main table of a MeasurementSet with its ANTENNA subtable. It selects all baselines for which the first station is a core station. The subquery returns a set containing the ids of the core stations, which is used to select the correct stations in the main table.

6.1.4 Join using derivedmscal

Several UDFs in the `derivedmscal` library make it possible to easily join a MeasuementSet or CASA Calibration Table with a subtable like ANTENNA or SPECTRAL_WINDOW. These functions know which columns to use making the join straightforward like in

```
SELECT mscal.ant1name(), mscal.ant2name() from ~/GER1.MS
```

The library also contain the more general SUBCOL function making it possible to join any table with a subtable. For example:

```
SELECT mscal.subcol('NAMES','NAME',NAMEID) from obs.parmdb
```

to get the parameter name for a LOFAR ParmDB table. A ParmDB table has a subtable NAMES containing the NAME and other info of a parameter. The column NAME_ID is used to reference that subtable.

6.2 Optimization

A lot of development work could 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.

- If the righthand side of the IN operator is a single value, IN is turned into ==.

- If the righthand side of the IN operator is a set of integer values with a min-max range of <=1024*1024, that set is turned into a boolean vector to get linear lookup time.

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 finding 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.

7 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.

7.1 UPDATE

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

updates all or some rows in the first table. More input tables can be given in the FROM clause and used in clauses like SET and WHERE. Unlike SQL it is possible to specify more tables in the UPDATE part which is the same as specifying them in the FROM clause. However, using the FROM clause makes it more clear that only the first table is updated.
`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 set DATA=t2.DATA, FLAG=t2.FLAG
           FROM that.ms t2 where all(FLAG)
UPDATE this.ms, that.ms t2 set DATA=t2.DATA, FLAG=t2.FLAG
                           where all(FLAG)
```

are equivalent. They copy the DATA and FLAG column of that.ms to this.ms for rows where all data in this.ms 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.

```
UPDATE vla.ms SET FLAG=F
```

It sets all elements of the arrays in column FLAG to False.

Type promotion and demotion will be done where 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
```

To solve this problem a temporary table (in this case in memory) can be used to save the value of e.g., ANTENNA1:

```
UPDATE my.ms
    set ANTENNA1 = ANTENNA2, ANTENNA2 = orig.ANTENNA1
    FROM [select ANTENNA1 from my.ms giving as memory] orig
```

7.1.1 Partial Array Update

It is possible to update part of an array using array indexing and slicing. E.g.,

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

The first example sets only a single array element, while the second one sets an entire row in the array. Similar to numpy it is also possible to use a mask like

```
UPDATE vla.ms SET FLAG[isnan(DATA)]=T
```

which sets the flag for the DATA values being a NaN. The data and mask must have the same shape. Note this is easier to write than the similar command

```
UPDATE vla.ms SET FLAG = iif(isnan(DATA), T, FLAG)
```

Masking and slicing can be combined making it possible to use masking on a part of an array. If the mask is given first, the slice is taken from both the data and mask. If the slice is given first, it is only applied to the data; the mask should have the same shape as the slice. For example:

```
UPDATE vla.ms SET FLAG[isnan(DATA)][,0]=T
UPDATE vla.ms SET FLAG[,0][isnan(DATA[,0])]=T
```

Both commands set the flag for NaN data in the XX polarization. The first one is somewhat easier to write, but processes the entire DATA and FLAG before taking the slice. The second one only reads and processes the required parts of DATA and FLAG, thus is more efficient.

7.1.2 Update columns from a masked array

If a column is updated with the value of a masked array, only the array part of the masked array is used. However, it is also possible to jointly update the data column and mask column from a masked array by combining them in parentheses like:

```
UPDATE vla.ms SET (DATA,FLAG)=maskedarray
```

It writes the data part into DATA and the mask into FLAG. As above it is possible to use a slice or mask operator on the combination like:

```
UPDATE vla.ms SET (DATA,FLAG)[,0]=maskedarray
UPDATE vla.ms SET (DATA,FLAG)[isnan(DATA)]=maskedarray
```

The slice or mask is applied to both columns.

7.2 INSERT

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

```
INSERT INTO table_list SET column=expr, column=expr, ...
INSERT INTO table_list [(column_list)] VALUES (exprlist),(exprlist),
       ... [LIMIT n]
INSERT INTO table_list [(column_list)] SELECT_command
```

The first form adds a single row setting the values in the same way as the UPDATE command. The second form is the SQL syntax and can add multiple rows. In this form the optional LIMIT part can also be given right after the INSERT keyword. In both forms it is possible to jointly specify data column and mask column if the value is a masked array. This is done by combining them in parentheses like `(DATA,FLAG)` as described in the previous subsection for the UPDATE command.

The first form adds one row to the table and puts the values given in the expressions into the columns.
For example:

```
INSERT INTO my.ms SET ANTENNA1=0, ANTENNA2=1
```

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

The second form can add multiple rows to the table. It puts the values given in the expression lists 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. Multiple expression lists can be given; each list results in the addition of a row (however, see LIMIT clause below).
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 in the new rows.
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),(2,3)
```

adds two rows, putting 0 and 2 in ANTENNA1 and 1 and 3 in ANTENNA2.

The LIMIT clause can be used to add multiple rows while giving fewer expressions. LIMIT can be given at the beginning or the end of the command. For example:

```
INSERT INTO my.ms (COL1) VALUES (rowid()) LIMIT 100
INSERT LIMIT 5 INTO my.ms (COL1,COL2) VALUES (0,0),(1,1)
```

The first example will add 100 rows where the value in each row is the row number. The second example shows that multiple expression lists can be given. It will iterate through them while adding rows. Thus COL1 and COL2 will have the values 0, 1, 0, 1, and 0 in the new rows.

The third form evaluates the SELECT command and adds 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. Column names and data types have to match, but their order can differ.
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.

7.3 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.

8 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 AS options colspecs LIMIT nrows DMINFO datamanagers
```

The command consists of 4 parts, all of them optional.

- The table name and options can be given in the same way as in the GIVING clause.
- The columns are defined in the `colspecs` part. If not given, a table without any column is created. Below column specification is described in more detail.
- An expression giving the number of rows can be specified in the LIMIT part. If not given, it defaults to 0.
- For expert users data managers can be defined in the optional DMINFO part described further down.

The CREATE TABLE command can be used in a nested query making it possible to fill it immediately. For example:

```
update [create table a.tab col1 int limit 10] set col1=rowid()
```

creates a table with one column and ten rows. The column is filled with the row number. Note that the following command would do the same.

```
select rowid() as col1 int limit 10 giving a.tab
```

8.1 Column specification

The `colspecs` part defines the column names, their data types, and optional shapes and units. It 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,...], DIRECT=0/1, 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, which is the default (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.
- `DIRECT=1` tells that a fixed shaped array column has to be stored directly. The default is 0. 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.

8.2 Data manager specification

The `datamanagers` 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. 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.

9 Modifying the table structure

TaQL can be used to modify the table structure, i.e., to add, rename, and remove columns and keywords. It is also possible to add rows. The syntax is:

```
ALTER TABLE tablename FROM table_list subcommand_list
```

It changes the table with the given name. The tables given in the optional FROM clause can be used in expressions defining keyword values. Any number of subcommands can be given, separated by whitespace and/or comma. The following subcommands can be given. They are explained in the next subsections.

```
ADD COLUMN colspecs DMINFO datamanagers
COPY COLUMN col TO new [AS [dtype] [tileshape=...]] [SET [expr]]
RENAME COLUMN old TO new, old TO new, ...
DELETE COLUMN column_list
```

```
SET KEYWORD name=value AS dtype, ...
COPY KEYWORD name=name AS dtype, ...
RENAME KEYWORD old TO new, old TO new
DELETE KEYWORD keyword_list
ADD ROW nrows
```

The nouns COLUMN and KEYWORD can also be given in the plural form. The whitespace between verb and noun is optional. For SQL-compatibility DROP can be used instead of DELETE.
For example:

```
ALTER TABLE my.tab RENAME COLUMN Col1 to Col1A, ADDCOLUMNS Col1 I4
```

renames column Col1 to Col1A and adds a new column Col1 with data type I4.

Note that TaQL has no way of showing keywords having a record value. The program *showtableinfo* can be used for that purpose.

9.1 ADD COLUMN

```
ADD COLUMN colspecs DMINFO datamanagers
```

adds one or more columns to the table. The specification of the columns and the optional data managers is the same as used in the CREATE TABLE command. Thus for each column a data type, dimensionality or shape, and unit can be given. The data manager(s) for the new columns can be specified in the DMINFO part. If not given, StandardStMan will be used. For example:

```
ADD COLUMN NCol1 R4, NCol2 R8 [UNIT="m", NDIM=3]
```

adds two columns, a 4-byte floating point scalar column and an 8-byte floating point 3-dim array column. They will be stored with StandardStMan.

9.2 COPY COLUMN

NOTE: This subcommand cannot be used yet and will give a parser error when used.

```
COPY COLUMN spec1, spec2, ...
```

makes it possible to copy columns, where each column specification looks like:

```
col TO new [AS [dtype] [tileshape=...]] [SET [expr]]
```

The new column gets the same description, but it is possible to define a new data type and/or tile shape. If the SET keyword is given values are written into the new column. If given, the value of an expression is stored in the new column, otherwise the contents of the input column. In its simplest form it copies a column adds one or more columns to the table. The specification of the columns and the optional data managers is the same as used in the CREATE TABLE command. Thus for each column a data type, dimensionality or shape, and unit can be given. The data manager(s) for the new columns can be specified in the DMINFO part. If not given, StandardStMan will be used. For example:

```
ADD COLUMN NCol1 R4, NCol2 R8 [UNIT="m", NDIM=3]
```

adds two columns, a 4-byte floating point scalar column and an 8-byte floating point 3-dim array column. They will be stored with StandardStMan.

9.3 RENAME COLUMN

```
RENAME COLUMN old1 TO new1, old2 TO new2, etc.
```

renames one or more columns in a table. For example:

```
RENAME COLUMN NAME to NAME_SAV, ADDR to ADDR_SAV
```

9.4 DELETE COLUMN

```
DELETE COLUMN col1, col2, etc.
```

removes one or more columns. Note that if multiple columns are combined in a TiledStMan, they have to be removed at the same time. Thus in that case

```
DELETE COLUMN col1, col2
DELETE COLUMN col1, DELETE COLUMN col2
```

are not the same, because the second example might fail.

9.5 SET KEYWORD

```
SET KEYWORD key1=value1 AS dtype, etc.
```

adds a keyword with the given value or replaces the value if the keyword already exists. The value of a keyword can be a scalar, array, or arbitrarily deeply nested record. See section 4.5 how to specify a keyword name in a column or nested record. The `AS dtype` part can be used to explicitly set the data type of a new keyword. For an existing keyword, the data type of the new value has to match the data type of the current value.

The value can be an expression, possibly using values from another table given in the FROM clause. It has to be a constant expression, thus cannot depend on column values. Of course, column values can be used when aggregated to a single value. If no data type is given, the data type of the expression result is used. If given, upward and downward coercion is possible (e.g., integer to float and also float to integer). For example:

```
SET KEYWORD key1=4
SET KEYWORD ::key1=4+5 AS U4
SET KEYWORD key1 = otherkey as I4
SET KEYWORD col::ckey.subrec.fld1 = [4,5,6.]
SET KEYWORD col::ckey=[=], col::ckey.subrec=[=]
SET KEWYORD key=[] AS I4
```

The 1st example sets table keyword *key1* to 4. Its data type is not given, thus is the expression's data type, in this case I8.
The 2nd example sets *key1* to 9, but as an unsigned 4 byte integer. Note that the `::` part is redundant.
The 3rd example copies the value of keyword *otherkey* while converting its data type to I4. Note that if no data type is given, the data type of *otherkey* is NOT preserved, because it is seen as a TaQL expression which has data type I8 (or R8).
The 4th example sets the *ckey.subrec.fld1* in column *col* to the given vector. It is a nested structure, thus field *fld1* in field *subrec* of column keyword *ckey* will be set. Its data type

will be R8.
Note that the command in the 4th example does not create the higher level records. If not existing yet, the 5th example can be used to create them, where [=] denotes an empty record (it is the old Glish syntax for an empty struct).
The last example shows how to create a key with an empty integer vector as value. In such a case the data type must be given, because it cannot be derived from the value.

Setting a keyword to the value of another keyword is easily possible. For instance:

```
SET KEYWORD key2 = otherkey
```

However, it has two problems.
1) As explained above the data type might not be preserved.
2) Keywords having a record value cannot be copied this way, because TaQL expressions do not support record values.

9.6 COPY KEYWORD

```
COPY KEYWORD key = otherkey AS dtype, etc.
```

copies the value of keyword *otherkey* to *key*. It can be used for any keyword value, thus also for records. The optional `AS dtype` part can be used to change the data type.

9.7 RENAME KEYWORD

```
RENAME KEYWORD old1 TO new1, old2 TO new2, etc.
```

renames one or more table or column keywords. If the old keyword is a field in a column or a nested record, the new name should only contain the new field name, not the full keyword path. For example:

```
RENAME KEYWORD NAME to NAME_SAV, Col1::CNAME to CNAME_SAV
RENAME KEYWORD KEYS.SET.NAME to NAME_SAV
```

The first example renames the table keyword NAME and the keyword CNAME of column Col1.
The second example renames a field in the nested records of table keyword KEYS.

9.8 DELETE KEYWORD

```
DELETE KEYWORD key1, key2, ...
```

removes one or more table or column keywords.

9.9 ADD ROW

adds the given number of rows to the table.

```
ADD ROW nrows
```

where *nrows* can be any expression. For example,

```
ALTER TABLE mytab ADD ROW [SELECT GCOUNT() from othertab]
```

makes *mytab* the same size as *othertab* (assuming it was empty).

10 Counting in a table

Before TaQL had the GROUPBY command, the COUNT command could be used instead of the gcount aggregate function to count the number of occurrences in a table. For backward compatibility this command can still be used, but its usage is discouraged, also because usually GROUPBY is faster.

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.

11 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 reduce function like SUM is used.

The CALC command was developed before the GROUPBY was available and before SELECT could be used without the FROM part. Currently, SELECT is more powerful than the CALC command. For example, multiple expressions can be given in a SELECT command. However, especially in Python sessions CALC has the advantage that it returns the results as a numpy-array or a list instead of a Casacore table.

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.

12 Examples

12.1 Selection examples

Some examples are given starting with simple ones.

12.1.1 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 some.ms WHERE ANTENNA1 != ANTENNA2`
selects the cross-correlations in a MeasurementSet.

`SELECT NAME FROM some.ms::ANTENNA`
selects the NAME of all antennae in a MeasurementSet.

`SELECT unique ANTENNA1,ANTENNA2 FROM some.ms`
gives the baselines used in a MeasurementSet.

`SELECT ANTENNA1,ANTENNA2 FROM some.ms GROUPBY ANTENNA1,ANTENNA2`
does the same using the GROUPBY clause.

`SELECT FROM mytable ORDERBY column0 DESC limit 10`
selects the 10 highest values of `column0`**.**

`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 NAME FROM some.ms::ANTENNA WHERE NAME ! p/[CR]S*/`
selects the names of the international LOFAR stations (not core or remote).

```
SELECT FROM some.ms WHERE ntrue(FLAG) >= 3
```
selects rows where at least 3 visibilities are flagged.

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

```
SELECT FROM some.ms WHERE any(ANTENNA1==[0,0,1] && ANTENNA2==[1,3,2])
```
selects the antenna pairs (baselines) 0-1, 0-3, and 1-2.
It requires some explanation. The two comparisons result in boolean vectors (with 3 elements). It matching elements are both true, the baseline in the table row matches. Thus the vectors are and-ed to see if any two matching elements are true.

```
SELECT FROM some.ms WHERE ANTENNA1 in [0,0,1] && ANTENNA2 in [1,3,2])
```
looks the same as above, but will select all baselines between the two sets, thus also 1-1, 1-3, and 0-2.

```
SELECT FROM some.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 direction (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 angdist([RA,DEC], [3h30m, 52deg]) <= 0d1m]
```
is another way to do the above.

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

```
SELECT ANTENNA1,ANTENNA2,sqrt(sumsqr(UVW[:2]))
        FROM some.ms GROUPBY ANTENNA1,ANTENNA2
```
finds the 10 longest baselines. It groups by ANTENNA1 and ANTENNA2 to get the unique baselines. UVW[:2] denotes the U and V coordinate giving the baseline length.

```
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. It uses a nested query to find the DATA_DESC_ID for each spectral window.

```
select from MY.MS where TIME in [select from ::SOURCE where REST_FREQUENCY < 180MHz
       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 VLA.MS,
       [select from VLA.MS where sumsqr(UVW[:2]) < 625] as TIMESEL
        where TIME in [select distinct TIME from TIMESEL]
        && any([ANTENNA1,ANTENNA2] in [select from TIMESEL giving
        [iif(UVW[2] < 0, ANTENNA1, ANTENNA2)]])
```

selects rows where an antenna (VLA has 25 m diameter) is shadowed.
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 (named TIMESEL) is done first otherwise two 2 equal selections are needed in the main WHERE command. The last line determines which antenna is shadowed (based on the W coordinate). The two lines above selects the times and baselines where an antenna is shadowed.

```
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+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.**

12.1.2 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.

12.2 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.

12.2.1 Applying running median to an image

The following command shows how a running median can be applied to an Casacore image.

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

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:

```
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 the image is updated, so it should have been copied before if the original data needs to be kept.

12.3 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.

12.4 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 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.

12.5 Aggregation/groupby examples

`select gcount(*) from my.ms`
counts the number of rows in the table.

`select TIME, gcount(*) from my.ms groupby TIME`
counts the number of rows (usually number of baselines) per time slot.

```
select ANTENNA1,ANTENNA2,gfirst(TIME),glast(TIME),gcount()
        from my.ms groupby ANTENNA1,ANTENNA2
```
counts the number of rows (usually number of time slots) and shows the first and last time per baseline.

```
select gmean(DATA) from my.ms
        groupby int((TIME - [select TIME from my.ms limit1][0]) / INTERVAL / 10))
```
calculates the average of DATA for every 10 time slots. Note it also averages in frequency and polarization. The following example shows how to average each frequency channel and polarization.

```
select boxedmean(gaggr(DATA), 10, 4) from my.ms
        groupby int((TIME - [select TIME from my.ms limit1][0]) / INTERVAL / 10))
```
calculates the average of DATA per polarization for every 10 time slots and 4 frequency channels. Note it it first combines the data of each 10 time slots in a single array, after which the `boxedmean` function is used to average every [10,4,1] box.

12.5.1 Obtaining the flux density from visibility data

The Miriad program `uvflux` estimates the source I flux density and its standard deviation at the phase center without having to make an image. A single, not too complicated TaQL command (courtesy Dijkema, Heald) provides the same functionality on a MeasurementSet. For LOFAR it is best to use baselines with a length between 5 and 10 km. The command shows various aspects of TaQL that are explained below. The numbers at the beginning of the lines point to the text following the example.

```
4.    select gstddev(SUMMED) as STDVALS,
4.           gmean(SUMMED) as MEANVALS,
4.           gcount(SUMMED) as NVALS
3.    from (select gmean(
1.                  sum(iif(FLAG[,0:4:3], 0, abs(DATA[,0:4:3])))
1.                   / nfalse(FLAG[,::3])
3.                 ) as SUMMED
           from ~/data/GER.MS
2.         where mscal.baseline('5km~10km) && !all(FLAG)
3.         groupby TIME)
```

A subquery is used to get the average flux (I = 0.5*(XX+YY)) per time slot.

1. **For each baseline it gets the mean of the channels. Note that it uses `sum/n` to ignore flagged visibilities. The `iif` function tells to use 0 for them. Also note that XX is the 1st and YY the 4th polarisation, hence [0:4:3] (or [::3]) indexes these polarisations. Once masked arrays are supported by TaQL, it could be written as: `mean(DATA[,::3][FLAG[,::3]])`**

2. **It only uses the baselines with lengths between 5 and 10 km where not all visibilities are flagged. Note that a `mscal` user defined function is used for the baseline selection as described in section 4.10.17.**

3. **Thereafter the average flux per time slot is determined in the subquery using the `gmean` aggregation and GROUPBY functionality. The result is an intermediate table with one column called SUMMED and a row per time slot.**

4. **Finally, the outer query uses aggregate functions to calculate the overall mean, standard deviation, and number of time slots from the result of the subquery. The final result is a table with 1 row and 3 columns.**

12.5.2 Number of fully flagged baselines per antenna

The example below counts per antenna the number of fully flagged baselines, excluding the autocorrelations. It uses grouping and aggregate functions twice; first per baseline, thereafter per antenna. It also uses the concatenation and backreferencing features. It also shows the results when using the `time` keyword, which shows that the processing time is dominated by the first query.

```
time select gsum(t1.cnt), t1.ANTENNA,
       (select NAME from ~/data/3C343.MS::ANTENNA)[t1.ANTENNA] as NAME
  from (select gcount() as cnt, ANTENNA1 as ANTENNA, ANTENNA2
          from ~/data/3C343.MS where all(FLAG) and ANTENNA1!=ANTENNA2
          groupby ANTENNA1,ANTENNA2) t0,
  [t0,
   (select cnt, ANTENNA2 as ANTENNA, 0 as ANTENNA2 from t0),
   (select 0 as cnt, rowid() as ANTENNA, 0 as ANTENNA2
           from ~/data/3C343.MS::ANTENNA)] t1
  groupby t1.ANTENNA orderby t1.ANTENNA

  From query          0.8 real        0.58 user        0.08 system
  From query            0 real           0 user           0 system
  From query            0 real           0 user           0 system
  Subquery              0 real           0 user           0 system
  Groupby               0 real           0 user           0 system
  Orderby               0 real           0 user           0 system
  Projection            0 real           0 user           0 system
 Total time          0.83 real         0.6 user        0.08 system
```

1. **The first inner select counts the number of fully flagged baselines per baseline and stores the result with the antennas making up a baseline in a temporary table with shorthand `t0`.. In this way the expensive counting part needs to be executed only once.**

2. **This table has to be summed for both antennas of the baselines, which is done in the second part creating a concatenated table with shorthand `t1`. It backreferences the first table `t0` twice. First to use it directly to count for ANTENNA1, thereafter to count for ANTENNA2 by doing a select to make the ANTENNA2 the ANTENNA column.**

3. **The table does not contain the antennas having no fully flagged baselines. Therefore the third part of the concatenation copies the ANTENNA table to insert zero counts for all antennas.**

4. **Finally the outer select (in the first line) sums the values in the concatenated table per antenna. It also retrieves the name of the antenna by indexing in the selection of all antenna names.**

13 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.

13.1 Python interface `python-casacore`

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.

```
import casacore.tables as pt
tab = pt.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 casacore.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 most generic 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.

```
import pyrap.tables as pt
t = pt.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.

13.2 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)')
```

13.3 Program `taql`

The program `taql` makes it possible to execute TaQL commands from the shell. Commands can be given in different ways:

- A TaQL command can be given directly as command line arguments to the *taql* program. The arguments will be combined to a single command (separated by spaces). Note that using multiple arguments instead of a single (quoted) argument makes it easier to use tab-completion for the table name. It will execute the command, show the result, and exit.

- **Using the -f option, the name of a file containing one or more TaQL commands can be given. The commands can be split over multiple lines, where a # can be used for comments. A semicolon has to be used to separate commands. This can be nested arbitrarily deep, thus a command file can execute another command file using a -f option.**

- **The program is run interactively if neither command nor -f is given. It will run until the user stops via the command `exit`, `quit`, or `q` or by giving Ctrl/D. Command editing and recall is possible, unless taql was built without readline support. Of course, a file can be used as input by redirection to stdin. This is more or less the same as using -f, but a command cannot be split over multiple lines and no semicolon is needed to separate commands. Interactive commands are kept in $HOME/.taql_history making command recall possible across multiple taql sessions.**

The commands can be given in a fully recursive way. For example, a command in an input file can invoke another TaQL command file using -f.
The following commands can be given:

- **`h`, `-h`, or `--help` shows brief help information about the *taql* program itself.**

- **`v` shows the *taql* version.**

- **`o` shows the current options settings (see below) on stderr.**

- **A full TaQL command. Note that the command `help` or `show` shows help info about the TaQL syntax and functionality.**

- **A full TaQL command preceded 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 `varname` can be used in subsequent commands such as `SELECT FROM $varname` or `SHOW TABLE $varname`.**
  If a command is not recognized, it is assumed an expression is given with an implicit SELECT. In this way it is easily possible to calculate expressions. Note that multiple expressions (separated by a comma) can be given, because it is the very same as selecting multiple column expressions in TaQL.

- **`?` shows the varnames.**

- **`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.

- **Comments can be given after the hash (#) character. Empty lines are ignored.**

A command can be preceded by zero or more options to specify if, how and where the results of a TaQL selection are printed. The options can be given at various levels:

- **The initial default options can be overridden by the options given when starting the *taql* program. These are global settings and used for all subsequent TaQL commands.**

- **A TaQL command can be preceded by options to override the global settings for that TaQL command only. However, when only giving options on a command line, the are persistent, thus get new global settings.**
- **The options usage in a TaQL command file behaves in a similar way. The initial option values are the settings of the level above which can be overridden temporarily or permanently by options on the command lines in the file. A nested command file inherits the settings from the level above. Note that persistent options settings do not go upwards.**

The output of a TaQL command can be printed. Most commands (such as UPDATE) will only show the expanded command and the number of rows affected. However, the CALC and SELECT command can also show the the results of the selected expressions. The result of a CALC command is always printed.
Selected columns in a SELECT command are optionally printed. If an implicit SELECT is done (thus if SELECT was added to the command) or if -ps is in effect, all results are printed. Otherwise if -pa is in effect, the first N rows are printed where N is defined by the -m option.

The following print options are available.

- `-ps` **or** `--printselect` **defines if all rows of selected columns will be printed. Note that all rows of an implicit SELECT are always printed.**
- `-pa` **or** `--printauto` **defines if auto printing is in effect meaning that only the first N rows of selected columns are printed. N is defined by the -m option.**
- `-pm` **or** `--printmeasure` **defines if values with a reference frame (such as times, positions, directions, ...) are pretty printed. Note that TaQL's** `str` **function offers very flexible pretty printing.**
- `-ph` **or** `--printheader` **defines if a header will be printed consisting of the names of the selected columns.**
- `-pc` **or** `--printcommand` **defines if the expanded TaQL command (as executed) will be printed.**
- `-pr` **or** `--printnrows` **defines if the number of rows handled by the command will be printed.**
- `-p` **or** `--printall` **defines all 5 print options (except -pa) above.**
- `-m` **defines the maximum number of rows to print if auto printing is used. It defaults to 50.**
- `-d separator` **defines the separator between printed columns. The default is a tab.**
- `-o filename` **defines the file in which to write the output.** ***stdout*** **and** ***stderr*** **are special names. Default is stdout.**

All -p options can be preceded by `no` **to negate settings.**
The initial default settings are '-nops -pa -ph -pm -nopc -pr -m 50'.
Note that an implicit SELECT always uses -noph -nopc -nopr regardless of their settings.
A few other options are available.

- **`-s style` or `--style style` defines the default TaQL style. It defaults to 'python'.**
- **`-h` or `--help` shows the help about the *taql* program (same as command `h`).**
- **`-v` or `--version` shows the *taql* version (same as command `v`).**
- **`-f filename` executes the TaQL commands in the given file.**

Note that '–' can be used to indicate the end of the options. This can be useful if the following TaQL command starts with a minus sign.

For example:

```
taql 'cos(pi())'
    -1
taql -d xx 1,2
    1xx2
taql -f t.cmd      # execute commands in file t.cmd
taql               # start interactive session
 o                 # show options
 help              # help about TaQL commands
 h                 # help about taql program
 -f t.cmd          # execute commands in file t.cmd
 cos(pi())
 unique TIME from an.ms              # implicit SELECT with printing
 -ps select unique TIME from an.ms   # explicit SELECT with printing
 select unique TIME from an.ms       # SELECT with auto printing
 -ps               # persistently set to print SELECT results
 select unique TIME from an.ms       # explicit SELECT with printing
 q                 # stop interactive session
```

13.4 C++ interface

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

13.4.1 TaQL query string

The function `tableCommand` in TableParse.h can be used to execute a TaQL command. The result is a TaQLResult object. Its function `isTable()` tells if the result contains a Table object or an TableExprNode object. The latter results from a CALC command. . E.g.,

```
TaQLResult result1 = tableCommand
       ("select from mytable where column1>0");
AlwaysAssertExit (result1.isTable());
Table seltab1 = result1.table();
Table seltab2 = tableCommand
       ("select column1,column2 from $1 where column2>5"
        " orderby time giving result.tab", seltab1).table();
```

These examples do the same as the Python 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 the Glish style is the default style.

13.4.2 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);
}
```

Class RecordGram can also be used to apply TaQL to C++ vectors of values or Records. The RecordGram class documentation and its test program describe these features in more detail.

13.4.3 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.

14 Writing user defined functions

A C++ user defined function has to be written as a class 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.

It is possible to write a UDF that operates on an individual expression (for each table row) and returns the result. It is, however, also possible to write a UDF acting as an aggregate function. In that case it will return a result based on the values of all rows in a group. See the desription of the GROUPBY clause for more information on the GROUPBY clause and aggregate 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.

A UDF class can contain a `HELP` function, which should return help information. This function is called by a help command like

```
show function meas [subtype]
```

It returns an overview of the functions in the UDF class and possible other information. The optional `subtype` argument can be used to return more specific information. Note that the same result is given by

```
meas.help('subtype')
```

TaQL finds a UDF by looking in a dictionary mapping 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.

14.1 UDFs in Python

NOTE: This section is for a future version of TaQL. It has not been fully implemented yet.

For performance reasons User Defined Functions will usually be implemented in C++. It is, however, possible to implement them in Python, both regular functions and aggregate functions. This can be done by means of the `pytaql` module of Casacore.

A UDF has to be implemented in Python by subclassing `pytaqlbase`, that can be imported from `Casacore.python`. The subclass has to implement a few functions, some are optional. The functions are called in the order given below.

- `__init__(self)`
  The class constructor must call the `__init__` function of the superclass.

- `needTable(self)`
  This optional function tells if the UDF needs the Table object of the table being queried. If True is returned, the function setTable is called thereafter. The Table object can be used by UDFs needing extra info (e.g., keywords) from the table being queried or from its subtables (comparable to derivedmscal). It requires the import of `pyrap.tables` at the beginning of the UDF script.

- `setTable(self, tab)`
  This function makes it possible to keep the Table object. It must be implemented if `needTable` returns True. The object should be kept like:

```
self.tab = pyrap.tables.table (tab, _oper=3)
```

- `setup(self, valuetypes, datatypes, units)`
  This function is called once at the beginning. It gets the value types, data types, and units of the function arguments. The length of the sequences tells the number of arguments.

```
valuetypes  int seq   value type of each argument
                      0=scalar 1=array 2=set
datatypes   int seq   data type of each argument
                      0=bool 1=int 2=float 3=complex
                      4=string 6=date
units       strings   unit of each argument (empty=no unit)
```

  The UDF should check if the argument types are correct and determine the result type. It has to return a dict containing the following fields:

```
ndim      int       dimensionality of result
                    -1=scalar 0=unknown
shape     int seq   shape (if fixed, otherwise empty sequence)
dtype     int       data type of result
unit      string    optional unit of result
isaggr    bool      True = UDF is aggregate function
```

- `getValue(self, argValues, rownr)`
  This function must return the function value for the given argument values. It is only called if `setup` does not set `isaggr=True`. Normally the `rownr` argument is not needed, but it could be used to obtain special info from the Table object for that row.

- `getAggrValue(self, rownrs)`
  This function must return the value of the aggregate function for the given rows. The argument values are not passed, because their sizes may exhaust memory. Instead the list of row numbers in the aggregation group are given. For each row the following function must be called to get a list of the argument values for that row.

```
getArgValues(self, rownr)
```

Such a UDF can be called in TaQL like `py.module.class` where the class defaults to the module name.
An example of UDFs in Python is given below. The first one is a regular UDF, the second one an aggregate UDF.

```
# tpytaql.py: Test script for pytaqlbase

from casacore.pytaqlbase import pytaqlbase
##import pyrap.tables as pt

class tpytaql(pytaqlbase):
    """
    A test (and example) for a pytaql UDF.
    It returns the sum of the values in the argument.
    """

    def __init__(self):
        """ The constructor must call the __init__ in the base class. """
        pytaqlbase.__init__(self)

    def setup(self, valuetypes, datatypes, units):
        """ Setup the pytaql object. """
        if len(valuetypes) != 1:
            raise ValueError("UDF tpytaql should have exactly 1 argument")
        self.isScalar = valuetypes[0]==0
        return {'ndim':0, 'dtype':datatypes[0], 'unit':units[0]}

    def getValue(self, argValues, rownr):
        """ Get the value of the function for the given table row. """
        if self.isScalar:
            return argValues[0];
        return argValues[0].sum()    # sum of numpy array



class tpytaqlaggr(pytaqlbase):
    """
    A test (and example) for a pytaql aggregation UDF.
    It returns the difference of the total of both arguments.
    """

    def __init__(self):
        """ The constructor must call the __init__ in the base class. """
        pytaqlbase.__init__(self)

## The following functions show how to get and keep a Table object.
## Note the import of pyrap.tables is also required.
##    def needTable(self):
##        return True
##    def setTable(self, tab):
##        self.tab = pt.table(tab, _oper=3)
##        print "nrows",self.tab.nrows()
```

```
def setup(self, valuetypes, datatypes, units):
    """ Setup the pytaql object. """
    if len(valuetypes) != 2:
        raise ValueError("tpytaqlaggr UDF should have exactly 2 arguments")
    self.isScalar0 = valuetypes[0]==0
    self.isScalar1 = valuetypes[1]==0
    return {'ndim':0, 'dtype':datatypes[0],
            'unit':units[0], 'isaggr':True}

def getAggrValue(self, rownrs):
    """ Get the value of the aggregate function for the given table rows. """
    v = 0;
    for rownr in rownrs:
        argValues = self.getArgValues (rownr);
        if self.isScalar0:
            v += argValues[0];
        else:
            v += argValues[0].sum()
        if self.isScalar1:
            v -= argValues[1];
        else:
            v -= argValues[1].sum()
    return v
```

15 Possible future developments

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

- **Add ROLLUP/CUBE to the GROUPBY clause.**
- **Implement the OVER/PARTITION clause.**
- **Add explicit JOIN clause (probably only equi-joins).**
- **Add UNION, INTERSECTION, and DIFFERENCE.**
- **Handle invalid subexpressions (e.g., exceeding array bounds) as null arrays which can be tested with the function ISNULL.**