picoSQL

Query language reference guide

Release 2.0.2

picoSQL - Query language reference manual

This manual presents descriptions of all SQL commands supported by picoSQL as well as syntax and description.
SQL supported by picoSQL comes from ANSI-92 e in particular from CAE (Common Application Environment) Specification "Structured Query Language" X/open of 1992. There are some limitations and some extensions that will be evidenced.

1) Syntax conventions

The following conventions are used in the SQL syntax description.

- All keywords are shown in upper case, while items that the user must replace with identifiers or expressions are shown in lower case. This is only a convention to better distinguish different parts of a command. picoSQL do not distinguish between upper and lower case, as much in the keywords as in the user's identifiers, so the command "SELECT MYTAB ..." is equals to "Select Mytab ..." and to "select mytab ...".
- Optional portions of a command are enclosed by square brackets [].
- Alternative options are separated by vertical bars.
- Curly braces enclose alternative options, separated by a vertical bar, between which only one must be chosen. **Warning!** Curly braces are also part of the language; they are used as escape sequences and in this case they must be written as they appear. In the continuation, the use of curly braces as escape sequences is evidenced to avoid ambiguities.
- Lists are shown with a list element followed by ", ...". This means that one or more list elements are allowed and if more than one is specified, the must be separated by commas.
- All the other characters, as parenthesis for example, are SQL syntax elements and must be written as they appear.

2) Language elements

In this paragraph the basic elements of SQL language are listed.

numbers A number is any sequence of digits followed by an optional decimal part and preceded by an optional negative sign. Exponential notation is allowed postplacing an 'E' and then an exponent. For example:

```
523
-43.001
7.8E5
3.4E-3
```

strings A string is any sequence of characters enclosed in single or double quotes (SQL standard allows only single quotes). A quote is represented inside the string by two adjacent quotes. The following strings are equivalent.

```
"L'opera"
'L''opera'
```

escape sequences Dates are not numbers nor strings, so an escape sequence is needed to identify a date type. picoSQL use curly braces to enclose these values. Allowed escape sequences are:

`{d '`*`YYYY-MM-DD`*`'}` to represent a date;
`{t '`*`hh:mm:ss.uuu`*`'}` to represent a time;
`{ts '`*`YYYY-MM-DD hh:mm:ss.uuu`*`'}` to represent a timestamp;

literal values Any string value, numeric value or date value is called literal value.

dynamic parameters When using picoSQL in a procedural language, some values of a query are often contained in a host variable. Because of this, you can prepare a query using the question mark character (?) instead of a literal value. In these cases of course, you need to supply the correct value before executing the query using an appropriate function call (SQLBindParameter).

identifiers An identifier is a name who identify a database element, like a table or a table attribute. The name can be any sequence of characters A through Z, a through z, 0 through 9 and underscore (_). The first character must be a letter.
picoSQL do not distinguish between upper and lower case. An identifier name cannot be a keyword.

separators A SQL command is a sequence of words separated by special characters, called separators. There are two kinds of separators. The first type is used only to separate the words and has no special meaning. Separators of this kind are the blank character (ASCII 32), tabulator (ASCII 9), carriage return (ASCII 13) and new line (ASCII 10).
The second type of separators has a special meaning. Separators of this kind are the following:

`, ( ) < > . = * + - / ?`

and the following characters couples:

`<> != >= <=`

3) Data types

PicoSQL manages the following data types.

CHAR(*dimension*)
VARCHAR(*dimension*)
Character data of maximum *dimension* size. The maximum size allowed is 32767. In the current release, there is no difference between these two types.

NUMERIC(*precision[, scale]*)
A decimal number with *precision* total digits and with *scale* of the digits after the decimal points. In the current release the maximum precision is 18.
The space required by an attribute of this kind can be computed by the following formula.

required-space = precision / 2 + 1

where the division is an integer division.

SERIAL	An integer 11 digits long, stored with the same format as a NUMERIC attribute. This number autoincrement itself every INSERT/REPLACE statement execution, starting from 1 and growing by 1. You can assign a predefined value to attributes of this type; if the assigned value is greater than any other in the table, this value became the reference for next insertions. However, a SERIAL attribute can be modified using an UPDATE statement.
SMALLINT	An integer that requires 2 bytes and can cantain values between -32767 and 32767. The number -32768 is interpreted as NULL. **Warning!** number of this type are stored in machine dependent format.
INTEGER INT	An integer that requires 4 bytes and can cantain values between -2147483647 e 2147483647. The number -2147483648 is interpreted as NULL. INTEGER and INT are synonyms. **Warning!** number of this type are stored in machine dependent format.
REAL	A single precision floating point number. It requires 4 bytes. **Warning!** number of this type are stored in machine dependent format. The smaller values (FLT_MIN) is interpreted as NULL.
DOUBLE	A double precision floating point number. It requires 8 bytes. **Warning!** number of this type are stored in machine dependent format. The smaller values (DBL_MIN) is interpreted as NULL.
DATE	A date. It requires 5 bytes. To specify a data you can use the appropriate escape sequence or also a string in the following format: *'YYYY-MM-DD'*
TIME	A time. It requires 5 bytes.To specify a time you can use the appropriate escape sequence or also a string in the following format: *'hh:mm:ss.uuu'*
TIMESTAMP	A timestamp. It requires 9 bytes. To specify a timestamp you can use the appropriate escape sequence or also a string in the following format: *'YYYY-MM-DD hh:mm:ss.uuu'*
BLOB	A binary large object whose size range from 0 to 2147483648 bytes. Allocation is done in block 1024 bytes long. picoSQL load the full blob in main memory, so you must sure to have enaugh free memory.
CLOB	This type is like the BLOB but it can contain only text data.

Each of these types can contain a conventional value, called NULL, that indicate that the value is unknown or not applicable to the context. When you insert a rows without specifying all values, for example, in the unspecified attributes go the NULL value.

4) Expressions

An expression is a single value that can also be computed using some operations. We call 'numeric expression' an expression whose result is a number. A numeric expression is formed from numeric literals, numeric dynamic parameters, numeric columns identifiers and numeric function results (see in the continuation). You can combine all this elements using the 4 arithmetic operators and get a new numeric expression. The allowed operators are, in the precedence order:

-	unary minus;
* /	multiplicatoin and division;
+ -	addition and subtraction.

Operataions with the same precedence are executed left to right. The execution precedence order can be altered using parenthesis.

We call 'string expression' an expression whose result is a string. A string expression is formed from string literals, string dynamic parameters, string columns identifiers and string function results (see in the continuation).

We call 'date/time expression' an expression whose result is a time, a data or a timestamp. A time expression is formed from escape sequences, dynamic parameters, time/data/timestamp columns identifiers and time/data/timestamp function results (see in the continuation).
In the current release, additions and subtractions between a number and a time expressions are allowed and the result is a new time expression equal to the original one plus/less the seconds specified by the number.

5) Functions

picoSQL has a group of built-in functions to return data from database. Function are of two kind: simple functions and aggregate functions.
Aggregate functions summarize data over a group of rows from database and return numeric, string or time values.
Simple functions execute computation on one ore more arguments and act on numeric, string or date/time values. Simple functions can be numeric functions, string functions or date/time functions.

5.1)Aggregate functions

COUNT ([DISTINCT] *column-name*) COUNT(*)	Return the number of rows in each group. If the DISTINCT *column-name* clause is specified, the function return the number of different values in the specified column.

MIN(*column-name*)	Return the minimum value found in the specified column of each group.
MAX(*column-name*)	Return the maximum value found in the specified column of each group.
SUM(*column-name*)	Return the total of the columns for each group. The specified column type must be numeric.
AVG(*column-name*)	Return the average of the columns for each group. The specified column type must be numeric.

☞ *picoSQL do not allow to use an aggregate function in an arithmetic expression.*

5.2 Numeric functions

In the following descriptions we denote a numeric expression as *num-expr* or *num-expr-1, num-expr-2* etc.

ABS(*num-expr*)	Returns the absolute value of the numeric expression.
ACOS(*num-expr*)	Returns the arc-cosine of the numeric expression.
ASIN(*num-expr*)	Returns the arc-sine of the numeric expression.
ATAN(*num-expr*)	Returns the arc-tangent of the numeric expression.
CEILING(*num-expr*)	Returns the smallest integer not less than the numeric expression.
COS(*num-expr*)	Returns the cosine of the numeric expression.
COT(*num-expr*)	Returns the cotangent of the numeric expression.
EXP(*num-expr*)	Returns the exponential function of the numeric expression.
FLOOR(*num-expr*)	Returns the largest integer not greater than the numeric expression.
LOG(*num-expr*)	Returns the natural logarithm of the numeric expression.
LOG10(*num-expr*)	Returns the logarithm base 10 of the numeric expression.
ROUND (*num-expr-1, num-expr-2*)	Rounds *num-expr-1* to *num-expr-2* places after the decimal point.
SIGN(*num-expr*)	Returns -1 if the numeric exprssion is less than 0, 1 if the numeric expression is greater than 0 or 0 if the numeric expression is equal to 0.
SIN(*num-expr*)	Returns the sine of the numeric expression.
SQRT(*num-expr*)	Returns the square roor of the numeric expression.
TAN(*num-expr*)	Returns the tangent of the numeric expression.
TRUNCATE (*num-expr-1, num-expr-2*)	Truncate *num-expr-1* at *num-expr*-2 places after the decimal point.

5.3) String functions

In the following descriptions, we denote a string expression as *string* or *string-1, string-2 etc.*

ASCII(*string*)	Returns a numeric value corresponding to the ASCII value of the first character of *string*..
CHAR(*num-expr)*	Returns a string 1 character long whose ASCII value is equal to *num-expr*.
CONCAT(*string-1, string-2*)	Concatenates two strings into one large string.
CONVERT(*num-expr*,SQL_CHAR)	Returns a string containing the representation of *num-expr*.
CONVERT(*string*,SQL_DOUBLE)	Converts a string in a number. If the value contained in *string* cannot be interpreted as a number, then this function returns 0.
CURRENT_TIMESTAMP(*format-string*)	Returns a string containing a timestamp of the current date and time. The result string has a format as specified in *format-string*. The following characters has a special meaning in *format-string*: `Y` year digit; `M` month digit; `D` day digit; `H` hour digit; `N` minute digit; `S` second digit; `T` millisecond digit (always 0). All the other characters are transcripted literally. You can get the standard timestamp format with the following *format-string*: `'YYYY-MM-DD HH:NN:SS.TTT'`
LCASE(s*tring*)	Converts all characters in *string* to lower case.
LOWER(s*tring*)	LCASE and LOWER are synonyms.
LENGTH(s*tring*)	Returns the length of *string*.
LOCATE(s*tring-1, string-2, num-expr*)	Returns the character offset (base 1) into the string *string-1* of the first occurrence of the string *string-2*, starting the search at the offset *num-expr*. If the string is not found, 0 is returned.
LTRIM(*string*)	Returns *string* with leading blanks removed.
RTRIM(*string*)	Returns *string* with trailing blanks removed.
SPACE(*num-expr*)	Returns a string *num-expr* blank characters long.
SUBSTRING(s*tring, num-expr-1, num-expr-2*)	Returns the substring of *string* starting at the given *num-expr-1* start position (origin 1) and *num-expr-2* characters long.
UCASE(s*tring*)	Converts all characters in *string* to upper case.
UPPER(s*tring*)	UCASE and UPPER are synonyms.

5.4) Date/time functions

The arguments of the following function must be of DATE type, TIME type or TIMESTAMP type. String expression are not allowed as arguments while escape sequences are allowed.

DAYOFMONTH(*timestamp*)	Returns a number from 1 to 31 corresponding to the month of the given timestamp.
HOUR(*timestamp*)	Returns a number from 0 to 23 corresponding to the hour of the given timestamp.
MILLISECOND(*timestamp*)	Returns a number from 0 to 999 corresponding to the milliseconds of the given timestamp.
MINUTE(*timestamp*)	Returns a number from 0 to 59 corresponding to the minute of the given timestamp.
MONTH(*timestamp*)	Returns a number from 1 to 12 corresponding to the month of the given timestamp.
NOW()	Returns a timestamp with the current date and time.
SECOND(*timestamp*)	Returns a number from 0 to 59 corresponding to the seconds of the given timestamp.
YEAR(*timestamp*)	Returns a number from 1 to 9999 corresponding to the year of the given timestamp.

6) Search conditions

Search conditions are formed from one or more predicates, combined one another using the logical operators AND, OR and NOT. Conditions are used as to choose a subset of the rows from one or more tables. A search condition is evaluated any time a rows is read and the rows is returned if and only if the result of condition is TRUE.

☞ *SQL standard use a three valued logic, so every condition evaluates as one of TRUE, FALSE or UNKNOWN. The truth tables of this kind of logic are the following:*

AND	*TRUE*	*FALSE*	*UNKN.*
TRUE	*TRUE*	*FALSE*	*UNKN.*
FALSE	*FALSE*	*FALSE*	*FALSE*
UNKN.	*UNKN.*	*FALSE*	*UNKN.*

OR	*TRUE*	*FALSO*	*UNKN.*
TRUE	*TRUE*	*TRUE*	*TRUE*
FALSE	*TRUE*	*FALSE*	*UNKN.*
UNKN.	*TRUE*	*UNKN.*	*UNKN.*

NOT	
TRUE	*FALSE*
FALSE	*TRUE*
UNKN.	*UNKN.*

picoSQL use a more simple boolean logic with only the TRUE and FALSE values. This behaviour implies only one difference between picoSQL and SQL standard: picoSQL evaluates TRUE an equality between two NULL values while standard SQL does not.

Two operands in a predicate must be of comparable types.

6.1)Comparison predicate

A comparison predicate is formed from 2 expression compared by a comparison operator as follows:

expression-1 comparison-operator expression-2

Comparison operators allowed are the following:

=	equal to
<> (or !=)	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

6.2)BETWEEN predicate

A BETWEEN predicate verifies if a value is contained between two values; the syntax is:

expression-1 [NOT] BETWEEN *expression-2* AND *expression-3*

This predicate, without the NOT operator, is equivalent to:

espressione-1 >= espressione-2 AND *espressione-1 <= espressione-3*

NOT operator reverses the meaning of the condition.

6.3)IN predicate

IN predicate compare an expression with a group of values; the syntax is:

expression [NOT] IN (*value-1*[,*value-2*] ...)

This predicate, without the NOT operator, is equivalent to:

expression = value-1 [OR *expression = value-2*] ...

NOT operator reverses the meaning of the condition.

6.4)LIKE predicate

LIKE predicate allows to search some string data using wild cards. The syntax is:

string-expression [NOT] LIKE *pattern*

The pattern may contain any number of wild cards. The wild cards are:

- underscore (_) match any one character;
- percent (%) match any string of 0 or more characters;
- back slash (\) prevent a wild card from having its special meaning;

For example, the pattern `'B%'` corresponds to any value starting with the B character, while the pattern `'B__'` corresponds any value starting with the B character and three characters long.

NOT operator reverses the meaning of the condition.

6.5)NULL predicate

This predicate verifies if an expression is null or not. The syntax is:

expression IS [NOT] NULL

NOT operator reverses the meaning of the condition.
In statndard SQL, this is the only way to verify if an expression has NULL value or not. PicoSQL instead returns TRUE also comparing two columns containing NULL value.

6.6)EXISTS predicate

This predicate allows to verify the existence at least a rows in any table that satisfies some conditions. The syntax is:

EXISTS (*subquery*)

The result of predicate is TRUE if the *subquery* result contains at least one row, FALSE otherwise.

7) Command list

This section includes an alphabetical listing of all SQL statements supported by the current release of picoSQL.

Syntax: Format 1:

ALTER TABLE *table-name* RENAME *new-table-name*

Format 2:

ALTER TABLE *table-name* ADD [COLUMN] *column-name data-definition*

Format 3:

ALTER TABLE *table-name* DROP [COLUMN] *column-name*

Purpose: Allows to modify a table definition.

See also: CREATE TABLE, DROP INDEX

Description: This command changes table structure without modifying the content.

Format 1 allows to change the table name.

Format 2 allows to add a new column to a table. The column is appended at the end of the row. If the table already has some rows, new column will be initialized to NULL. Column name must be univoque in the table. COLUMN clause has no effects.

Format 3 allows to drop a column from a table. The column to drop can contain data and in such case, this data are lost. The column to drop cannot be part of an index; in this case you need to drop the index before and then the column.

Syntax: CALL *procedure-name* ([*argument*[, *argument*]...])

Purpose: Allows to invoke a stored procedure

See also: SELECT

Description: This statement allows the invocation of a stored procedure previously created. The argument number can be fixed or variable, depending on the procedure declaration. Arguments are expressions, as in a SELECT statement.
Like any other SQL statement, CALL statement returns only a success or error code; it can generate a result set, like a SELECT statement, or it can generate nothing, like statements INSERT, UPDATE and DELETE. Introduced from release 2.0.

Syntax: CREATE [UNIQUE] INDEX *index-name*
ON *table-name*
(*column-name* [ASC|DESC] [, *column-name* [ASC|DESC]] ...)

Purpose: Allows the creation of an index on the specified table. Indexes are important to improve performances.

See also: DROP

Description: This statement creates an index on the specified columns of the specified table. Indexes are automatically used by picoSQL to improve performances in the searching and when an ordering is selected using the ORDER BY clause.

An index can be created on a table in any time, also if the table is not empty.

The UNIQUE clause ensures that there will not be two rows in the table with the same values in all the columns of the index.

The *index-name* is not used by picoSQL, but it must be specified for compatibility with SQL standard. Every index has a conventional name, formed in the following way:

table-name_progressive-number

The progressive number start from 02 because number 01 is reserved to primary key. The index conventional name must be used in the DROP INDEX statement to drop an existant index.
If you drop the *n*th index, the *(n+1)*th index becomes the *n*th and so on.

An index can be formed from 8 parts, and a part can be formed from more than one adjacent columns of non-native type (CHAR, VARCHAR, NUMERIC, DATE, TIMESTAMP). An index can be altogether 255 characters long.

The DESC clause is not implemented in the current release.

Syntax: CREATE TABLE *table-name*
(*column-name data-definition* [PRIMARY KEY]
[, *column-name data-definition* [PRIMARY KEY]] ...)
[, PRIMARY KEY (*column-name* [,*column-name*]...)]

Purpose: Allows the creation of a database table.

See also: DROP, CREATE INDEX

Description: This statement create a new table on the database.

The PRIMARY KEY clause, specified after a column declaration, cause this column to become a unique index on the table. Differently from the SQL standard, this clause can be part of more than one columns, creating in such way a primary index composed from more than one columns.

The PRIMARY KEY clause, specified after the declaration of all the columns, allows to specify a primary index composed by more than one columns in any order.
It is not allowed to use both the formats.

Syntax: CREATE VIEW *view-name* [(*column-name* [, *column-name*] ...)]
AS *simple-query* [WITH CHECK OPTION]

Purpose: Allows the creation of a logical view.

See also: DROP, SELECT

Description: This statement creates a new logical view on the database. A logical view is useful to give a different perspective on the data even though it is not stored in that way. Practically, a logical view is a query, made using a SELECT statement, that is stored and can be used as a real table. However the view do not exists phisically, so it is newly computed for any query. The SELECT statement cannot have aggregate functions nor ORDER BY or GROUP BY clauses.

A logical view can inherit the columns name from the names of the original columns or you can specify new names during the creation; in the latter case the number of specified names must correspond to the number of column returned by the query.

Creating a view that join two or more tables, picoSQL requires the join condition expressed with the appropriate *join-clause* (see the SELECT statement).

A view without joins can be update like any other table. If the clause WITH CHECK OPTION is specified, any update or insert to the view is rejected if do not meet the criteria of the view defined by its SELECT statement.

Syntax: DELETE FROM *table-name*
[WHERE {*search-condition* |CURRENT OF *cursor-name*}]

Purpose: Allows to delete an arbitrary number of rows from a table.

See also: INSERT, UPDATE, SELECT

Description: This statement deletes all the rows from the named table that satisfy the search condition. If the WHERE clause is not specified, all the rows from the table are deleted. The CURRENT OF clause can be used, instead of the search condition, to delete only the current row read by a cursor. However, this clause can be used only from a program language because a cursor name can be gotten or set only calling an API (SQLGetCursorName and SQLSetCursorName respectively).

Syntax: DROP { INDEX *index-name* | TABLE *table-name* | VIEW *view-name* }

Purpose: Allows to drop an index, a table or a logical view.

See also: CREATE INDEX, CREATE TABLE, CREATE VIEW

Description: This statement drop an index, a table or a view from the database.

Dropping an index, the freed space remains allocated and it is use for the remaining indexes or for new indexes.
Dropping a table, the freed space is immediately released to the operating system and can be used by other application too.
Dropping a view, the small freed space can be reused only for creating other views.

Syntax: Format 1:

INSERT INTO *table-name* [*(column-name* [, *column-name*]...)]
VALUES (*expression* |NULL [, *expression* | NULL] ...)

Format 2:

INSERT INTO *table-name* [*(column-name* [, *column-name*]...)] *query*

Purpose: Allows the insertion of one or more rows in a table.

See also: REPLACE, DELETE, SELECT

Description: This command allows to insert one o more rows in the named table.
Format 1 allows to insert only one row with the specified values after the VALUES clause. If the optional column names list is specified, expressions after the VALUES clause are in positional correspondance with the columns name (the first with the first, the second with the second and so on). If the columns list does not comprise all the attribute defined in the table, the value NULL is put in the missing attributes.
If the optional column names list is not specified, all the attributes of the table must be specified after the VALUES clause: the corrispondence between the table attributes and the experssion is done by position. The row is inserted in an arbitrary position.

Format 2 allows to get the expression to insert from a SELECT statement. Also in this case there is positional correspondance between specified attributes, or table attributes, and the result expressions from the query.

If one or more attributes of the table are declared as SERIAL type and they contain the NULL value, they will receive a sequential number. Assigned values can be obtained using the SELECT SERIAL statement (introduced from release 2.0) on condition to use the same statement handle used for INSERT statement.

Syntax: Format 1:

REPLACE INTO *table-name* [*(column-name* [, *column-name*]...)]
VALUES (*expression* |NULL [, *expression* | NULL] ...)

Format 2:

REPLACE INTO *table-name* [*(column-name* [, *column-name*]...)] *query*

Purpose: Allows to insert or rewrite one or more rows in the named table.

See also: INSERT, DELETE

Description: This command is simular to the INSERT statement, the difference is that if the row insertion cannot be done because of one (and only one) unique index duplication, REPLACE rewrite the full row. From a logical point of view REPLACE is equivalent to execute a DELETE statement based on the first unique index and then an INSERT statement. However, the REPLACE command is atomic and much more efficient than executin DELETE and INSERT statement. You can get a duplicate index error only if the table has more than one unique index and if the REPLACE cause the duplication of an index successive the first one.

Syntax: Format 1:

SELECT [ALL | DISTINCT] *expression-list*
FROM { *table-name* [[AS] *alias*][, *table-name* [[AS] *alias*]] ... |
join-clause }
[WHERE *search-condition*]
[GROUP BY *column-name* [, *column-name*] ...]
[HAVING *search-condition*]
[UNION *select-statement*]
[ORDER BY {*integer* [ASC|DESC]|*expression* [ASC | DESC]}
[, {*integer* [ASC|DESC]|*expression* [ASC | DESC]}]...]
[LIMIT *integer*]
[OFFSET *integer*]

Format 2:

SELECT *expression-list*
FROM *table-name*
[WHERE *search-condition*]
FOR UPDATE

Format 3 (release >= 2.0):

SELECT SERIAL

Purpose: Allows to query the database.

Vedi anche: CREATE VIEW, UNION, INSERT

Descrizione: This statement is the more complex one because it is the only one that allows to query the database.
For format 3 use, see the INSERT statement.
Format 2 is a subset of format 1 with the FOR UPDATE clause and it is used to lock the read row. The row is unlocked when the next row is read or the cursor is closed. The format 1 clauses are the following.

ALL | DISTINCT
If the DISTINCT clause is specified, duplicate output rows are eliminated. This is called 'projection' of the result of the command. Pay attention that to eliminate the duplicate rows, picoSQL execute a sort of the output rows and this operation can take significantly longer to execute, especially the number of retrived rows is very high.
The ALL clause instead returns all the rows and this also the default behaviour.

expression-list
expression-list specifies what will be retrived from the database. It has the following form:

{ * | *expression* [[AS] *alias-name*] [, *expression* [[AS] *alias-name*]]... }

If asterisk (*) is specified, it is expanded to select all columns of all tables in the FROM clause. Aggregate functions are allowed in the expression list.

Alias names can be used throughout the query to represent the aliased expression. Alias names are usually displayed by interactive programs, at the top of each column of output from the SELECT statement.

FROM

Specifies the tables list from which the data are retrieved. When data are retrieved from more than one table, usually one or more join conditions must be specified. A join reduces the result set based on a condition that bind a column from one table to a column on another table. PicoSQL allows two methods to specify a join:

- you can put the join conditions together the search conditions after the WHERE clause (see in the continuation) ;
- you can specify the join condition in the FROM clause directly with an appropriate syntax.

If you use the former method, the FROM clause is only a list of table names. Alias names are necessary to distinguish between table instances when referencing the same table more than once in the same query (self joins).

The latter method is more complex, but it distinguish the join conditions from the search conditions and it allows the outer joins. The *join-clause* syntax is the following:

table-name [[AS] *alias*] {INNER | LEFT OUTER| RIGHT OUTER}
JOIN {*table-name* [[AS] *alias*] | *join-clause*}
ON *join-condition*

This syntax allows to specify any number of joins, also of different types, by nesting the *join-clause* one in another.

If you put the join conditions after the WHERE clause or you use the join clause with the INNER JOIN option, you get the same results but performances may vary. Infact, in the former case picoSQL choose the optimal scanning table order to minimize the number of reads, in the latter case the scanning tables order is specified by the join clause itself.

WHERE

Specifies a *search-condition* that restrict the rows that will be selected from the tables. Also the join conditions can be specified in this clause. See "Search conditions".

GROUP BY

Group multiple rows together from the database. GROUP BY expressions must also appear in the select list. The result of the quesry contains one row for each distinct set of values in the named columns. Aggregate functions can then be applied to these groups to get meaningful results.

HAVING

Restricts which groups will be selected based on the group values and not on the individual rows values. The HAVING clause can only be used if the command has a

GROUP BY clause.

ORDER BY
Sort the results of a query. Each item in the ORDER BY list can be labeled as ASC for ascending order or DESC for descending order. Ascending is assumed if neither is specified. If the expression is an integer N, then the query results will be sorted by the N'th itm in the select list.

LIMIT
Limits the number of retrieved rows to the number specified as argument. If the total number of selected rows is less than the number specified as argument, this clause has no effects.

OFFSET
Specifies the offset of the first row to return. If the total number of selected rows is less than the number specified as argument, the result is a no data found error.

Syntax: *select-without-order-by* UNION [ALL] *select- without-order-by*
[UNION [ALL] *select- without-order-by*] ...
[ORDER BY *integer* [ASC | DESC][, *integer* [ASC | DESC]] ...]

Purpose: To combine the results of two or more select statements.

See alse: SELECT

Description: The results of several SELECT commands can be combined into a larger result using UNION. The component SELECT commands must each have the same number of items of the same type in the select list and cannot contain an ORDER BY clause. UNION statement eliminates duplicate rows: the ALL clause retains duplicate rows. The ORDER BY clause is analogous to that one used in the SELECT statement but allows to specify only integer numbers that specify the position of the columns to be sorted.

Syntax: UPDATE *table-name*
SET *column-name* = {*expression* | NULL}
[, *column-name* = {*expression* | NULL}] ...
[WHERE {*search-condition*|CURRENT OF *cursor-name*}]

Purpose: To modify one or more rows of a table.

See also: INSERT, DELETE, SELECT

Description: This command is used to modify rows of one table. Each named column is set to the value of the expression (or NULL) on the right side of the equal sign. Even *column-name* can be used in the expression, the old value will be used.
WHERE clause allows to modify only those rows which satisfy the search condition. If the WHERE clause is not specified, all the rows in the table will be updated. The CURRENT OF clause can be used, instead of the search condition, to update only the current row read by a cursor. However, this clause can be used only from a program language because a cursor name can be gotten or set only calling an API (SQLGetCursorName and SQLSetCursorName respectively).

8) Query optimizer

picoSQL use a query optimizer which find the best way to satisfy the required queries. Only two read startegies are implemented, the sequential scanning of the whole table and the use of an appropriate index.
picoSQL find the best startegy analyzing the search conditions, so it is very important to define appropriate indexes and to use search condition that can use them, especially on large tables.
To show, in a simple way, how picoSQL works, we suppose now to have a table with only one index of a single column, whose name is I.

The query optimizer analizes the search condition only in a *formal* way, that is without taking in consideration the effective values neither the values really stored in the table. To understand why it acts in this way, it must consider that often the queries contain dynamic parameter, as in the following example:

```
I = ?
```

This query can be executed more than one times, substituting a different value in place of the question mark. Clearly in such case, it is more efficient to analize the query only one time and to find a searching strategy that is good for any parameter value.
To repeat the analisys each time the value change is more expensive and usually do not improve the strategy.
On this base, picoSQL is able to find a good search index, if one exists.

To use an index for searching some data, the search condition must characterize one or more *exhaustive intervals* on the columns of the index, containing all the required rows, as in the following example:

```
I < 5
```

Because the query optimizer do not consider the data on the table, sometimes the use of an index do not improve the performances: for example, if the column I contains only values that are less than 5, the previous search condition do not limit the number of required rows, so a sequential scanning would be more efficient.

To put any condition in AND to a condition that characterize an exhaustive interval, do not influence the goodness of an index, so, from the optimizer point of view, the following condition is as good as the previous one.

```
I < 5 AND A > 10
```

Things are different changing the AND operator with the OR operator. In the following condition:

```
I < 5 OR A > 10
```

the condition on the index do not characterize all the rows required, so the interval is not exhaustive and the use of the index is not useful. To have an index defined on the A columns do not change the situation because not even A characterize an exhaustive interval. To read both the indexes, eliminating the duplicate rows, is expensive and can cause to scanning the whole table two times in the worst case. So, in such situations, the optimizer prefers to scan the whole table one time.

From this observation, we can derive a simple rule of thumb: if you use the OR operator, probably the whole table must be scanned. However, this is not true for any query: in some cases the optimizer is able to find exhaustive intervals also if there are OR operator in the search condition, as in the following example:

```
I < 5 OR I > 10
```

In this case there are two intervals, but they contain all the required rows. picoSQL is able to take advantage of indexes when there are more than one exhaustive interval, also if they are overlapped, as in the following case:

```
I > 5 OR I > 10
```

When an index is composed by more than one column, the theory is the same but to understand if the optimizer can take advantage of the use of an index is more difficult.
The optimzer can take advantage of the *n*th part of a composed index if $n = 1$ or if the optimizer can take advantage of the $(n - 1)$th part. We suppose now to have a composed index on the columns named I, J and K. Lets look the following search conditions.

```
I > 5 AND J < 10 AND K = 4
I > 5 AND K = 4
K = 4
```

In the first search condition the optimizer can take advantage of all parts of the index, in the second one the optimizer can take advantage of only the first part of the index while in the third one the optimizer cannot use the index at all.
Now we can combine this new rule with the previous one to understand if an index can be used or not. Lets look the following search conditions.

```
(I = 5 AND J = 10 AND K = 4) OR (I = 6 AND J = 11 AND K = 5)
(I = 5 AND J = 10 AND K > 4) OR (I = 5 AND J > 10) OR I > 5
```

In the first one the optimizer can take advantage of all three part of the index while in the second one the optimizer can take advantage of only the part on column I; in fact in this case J and K can have any values, so they cannot used as indexes. In thruth, if we analyze the values, we can characterize an exhaustive interval, but, to do this, we need to considers the cotextual values and picoSQL do not do this.