picoSQL

Guida di riferimento del linguaggio di interrogazione

Release 2.0.2

Copyright © 2003 Picosoft s.r.l. - Corso Italia 178 - Pisa Italy
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation.
The GNU Free Documentation License is available from www.gnu.org.

picoSQL - Guida di riferimento al linguaggio SQL

Questo manuale illustra tutti i comandi SQL supportati da picoSQL con la relativa sintassi e una descrizione.
Lo SQL supportato da picoSQL si rifà allo standard ANSI-92 e in particolare alla CAE (Common Application Environment) Specification "Structured Query Language" X/open del 1992. Rispetto agli standard suddetti ci sono alcune limitazioni e alcune estensioni che saranno evidenziate nel testo.

1) Convenzioni sintattiche

In questo manuale sono adottate le seguenti convenzioni per la descrizione della sintassi SQL.

- Tutte le parole riservate sono scritte in lettere maiuscole, mentre gli elementi che l'utente deve rimpiazzare con identificatori o espressioni sono scritti in lettere minuscole. Questa è solo una convenzione per distinguere meglio i vari elementi che compongono un comando. picoSQL non fa distinzione tra maiuscoli e minuscoli, sia nelle parole riservate che nei nomi utente, per cui il comando "SELECT PIPPO ..." è equivalente a "Select Pippo ..." e a "select pippo ...".
- Porzioni di comando opzionali vengono racchiuse tra parentesi quadre [].
- Le barre verticali separano opzioni mutuamente esclusive.
- Le parentesi graffe {} racchiudono opzioni diverse, separate da una barra verticale, tra le quali una deve essere scelta. **N.B.** Le parentesi graffe vengono usate nel linguaggio come sequenze di escape (vedi nel seguito) per cui in tal caso debbono essere riportate come appaiono. Nel seguito è sempre rimarcato l'uso delle graffe come sequenza di escape in modo da evitare ambiguità.
- I puntini (...) indicano che la sintassi racchiusa nelle parentesi quadre o graffe precedenti può essere ripetuta un numero indefinito di volte.
- Tutti gli altri caratteri, come le parentesi tonde, sono elementi sintattici dell'SQL e debbono essere riportati come appaiono.

2) Elementi del linguaggio

Riportiamo in questo paragrafo una lista degli elementi basilari del linguaggio SQL.

numeri Un numero è una sequenza di cifre seguita opzionalmente da una parte decimale e opzionalmente preceduto da un segno negativo. Sono ammessi numeri in notazione esponenziale a base 10 posponendo la lettera E seguita dal valore dell'esponente. Sono numeri validi, per esempio:

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

stringhe Una stringa è una qualsiasi sequenza di caratteri racchiusa tra apici singoli o doppi (SQL standard prevede solo apici singoli). Per inserire il carattere usato per racchiudere la stringa nella stringa stessa è sufficiente ripeterlo due volte. Le seguenti stringhe sono, per esempio, equivalenti:

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

sequenze di escape Nel caso si voglia indicare dei valori non ascrivibili direttamente al gruppo dei numeri o delle stringhe, come per esempio le date, si può utilizzare una sequenza di escape che identifica il tipo in modo univoco. picoSQL usa le parentesi graffe per racchiudere questi valori. Le sequenze di escape riconosciute da picoSQL sono le seguenti:

`{d '`*`AAAA-MM-GG`*`'}` per indicare una data;
`{t '`*`oo:mm:ss.uuu`*`'}` per indicare un'ora del giorno;
`{ts '`*`AAAA-MM-GG oo:mm:ss.uuu`*`'}` per indicare un'ora di un determinato giorno (timestamp);

valori letterali Un qualsiasi valore stringa, numerico o definito con una sequenza di escape è detto valore letterale (literal).

parametri dinamici Se si usa picoSQL da un linguaggio di programmazione, può accadere che certi valori di una query siano variabili e dipendenti dal contesto. Per questo motivo si può preparare una query al database usando il carattere punto interrogativo (?) al posto di un valore letterale. Ovviamente in questi casi è necessario fornire al database il valore voluto prima di eseguire la query tramite un'apposita funzione (SQLBindParameter).

identificatori Un identificatore è un nome che identifica un elemento del database, come una tabella o un attributo di una tabella. Il nome deve iniziare con una lettera e può contenere lettere, numeri e caratteri di sottolineatura (_). Per picoSQL negli indentificatori le lettere minuscole sono equivalenti alle relative lettere maiuscole. Un identificatore non può coincidere col nome di una parola riservata.

separatori Un comando SQL è formato da un certo numero di parole separate da caratteri particolari detti separatori. Un separatore può avere l'unico scopo di separare le parole oppure avere associato anche un altro significato. Fanno parte del primo tipo i caratteri spazio (ASCII 32), tabulazione (ASCII 9), ritorno carrello (ASCII 13) e nuova linea (ASCII 10). Fanno parte del secondo tipo i caratteri:

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

e le coppie di caratteri:

```
<> != >= <=
```

3) Tipi dato

In picoSQL esistono i seguenti tipi di dato.

CHAR(*dimensione*) VARCHAR(*dimensione*)	Definisce un attributo stringa di caratteri lunga al massimo *dimensione* caratteri. Nella release attuale di picoSQL non c'è differenza tra i due tipi in quanto entrambi sono a lunghezza fissa e quindi corrispondenti al tipo CHAR nello standard SQL. La lunghezza può arrivare fino 32767 caratteri, che é anche la massima dimensione di una riga di tabella, mentre lo standard SQL prevede una lunghezza massima di 255 caratteri.
NUMERIC(*precisione[, decimali]*)	Definisce un attributo numerico a virgola fissa che può contenere fino a *precisione* cifre, di cui *decimali* per la parte decimale. Nella versione attuale la massima precisione ammessa è 18. Lo spazio occupato da ciascun attributo di questo tipo si calcola con la seguente formula *spazio-occupato = precisione / 2 + 1* dove la divisione si considera intera.
SERIAL	Definisce un attributo numerico intero di 11 cifre, memorizzato nello stesso formato di un NUMERIC, che si incrementa di 1 automaticamente a ogni inserimento, partendo dal valore 1. In inserimento è concesso assegnare a colonne di questo tipo un valore predeterminato; nel caso che tale valore sia maggiore di quelli già esistenti, esso diventa il riferimento per i successivi inserimenti. Una colonna SERIAL può però venir modificata a piacere con un'operazione di aggiornamento (UPDATE) .
SMALLINT	Definisce un attributo numerico intero di 2 byte il cui valore è compreso tra -32767 e 32767. Il valore -32768 viene considerato come NULL. **N.B.** i dati di questo tipo sono memorizzati nel formato nativo del computer ospite.
INTEGER INT	Definisce un attributo numerico intero di 4 byte il cui valore è compreso tra -2147483647 e 2147483647. Il valore -2147483648 viene considerato come NULL. INTEGER e INT sono sinonimi. **N.B.** i dati di questo tipo sono memorizzati nel formato nativo del computer ospite.
REAL	Definisce un attributo numerico in floating point di 4 byte. **N.B.** i dati di questo tipo sono memorizzati nel formato nativo (float) del computer ospite per cui i valori minimo e massimo dipendono da quest'ultimo. Il valore più piccolo (FLT_MIN) viene considerato come NULL.
DOUBLE	Definisce un attributo numerico in floating point di 8 byte. **N.B.** i dati di questo tipo sono memorizzati nel formato nativo (double) del computer ospite per cui i valori minimo e massimo dipendono da quest'ultimo. Il valore più piccolo (DBL_MIN) viene considerato come NULL.

DATE	Definisce un attributo che contiene una data. Occupa 5 byte. Per inviare un dato di questo tipo al database, è necessaria una stringa col seguente formato: *'AAAA-MM-GG'* oppure la corretta sequenza di escape.
TIME	Definisce un attributo che contiene un'ora con approssimazione al millesimo di secondo. Occupa 5 byte. Per inviare un dato di questo tipo al database, è necessaria una stringa col seguente formato: *'oo:mm:ss.uuu'* oppure una adeguata sequenza di escape.
TIMESTAMP	Definisce un attributo che contiene una data e un'ora con approssimazione al millesimo di secondo. Occupa 9 byte. Per inviare un dato di questo tipo al database, è necessaria una stringa col seguente formato: *'AAAA-MM-GG oo:mm:ss.uuu'* oppure una adeguata sequenza di escape.
BLOB	Definisce un insieme di byte che occupa uno spazio variabile compreso tra 0 e 2147483648 byte. L'allocazione viene fatta in blocchi di 1024 byte. Quando picoSQL legge un attributo di questo tipo, lo carica in memoria centrale, per cui bisogna essere sicuri di averne a sufficienza.
CLOB	Come il BLOB ma è destinato a contenere testi.

Ciascuno di questi tipi può contenere un valore convenzionale, chiamato NULL, che indica che il valore è sconosciuto o non applicabile al contesto. Per esempio, quando si inserisce una riga non specificando il valore di alcuni attributi, in questi viene posto il valore NULL.

4) Espressioni

Un'espressione rappresenta un singolo valore che può essere ottenuto anche tramite un'operazione.
Un'espressione viene definita numerica se il valore che restituisce è un numero. Possono far parte dell'espressione numerica i valori letterali numerici, i parametri dinamici, gli identifiatori di colonna e i risultati delle funzioni (vedi nel seguito) che restituiscono valori numerici. Un'espressione numerica può essere ottenuta combinando questi elementi tramite i 4 operatori aritmetici.
Gli operatori consentiti sono, in ordine di precedenza dalla massima alla minima, i seguenti:

-	meno unario;
* /	moltiplicazione e divisione;
+ -	addizione e sottrazione.

Le operazioni con uguale precedenza sono eseguite da sinistra a destra. L'ordine di precedenza dell'esecuzione delle operazioni può essere alterato tramite l'utilizzo di parentesi tonde `()`.
Un'espressione viene definita di tipo stringa se il valore che restituisce è una stringa. Un'espressione stringa è formata o da un valore letterale stringa, o da un parametro dinamico, o da un identificatore

di colonna, o infine dal risultato di una funzione stringa (vedi nel seguito).
Un'espressione viene definita temporale se restituisce una data, un tempo o un timestamp. Un'espressione temporale è formata o da una sequenza di escape, o da un parametro dinamico, o da un identificatore di colonna. Nella release attuale sono permesse addizioni e sottrazioni tra un numero e un'espressione temporale e il risultato è un'espressione temporale pari a quella di partenza più/meno i secondi specificati dal numero.

5) Funzioni

picoSQL ha predefinito una serie di funzioni al proprio interno per rendere più significativi i risultati delle ricerche sul database. Queste funzioni sono principalmente di due tipi: funzioni semplici e funzioni aggreganti.
Le funzioni aggreganti vengono applicate sull'insieme dei risultati di una colonna in una ricerca e restituiscono un valore che può essere numerico o stringa.
Le funzioni semplici eseguono dei calcoli su uno o più argomenti e restituiscono il risultato che può essere numerico o stringa. Le funzioni semplici possono essere a propria volta distinte in funzioni numeriche, funzioni stringa e funzioni temporali a seconda del tipo di argomento su cui operano.

5.1)Funzioni aggreganti

COUNT ([DISTINCT]*nome-colonna*) COUNT(*)	Restituisce il numero di righe che hanno soddisfatto la ricerca. Nel caso sia specificato anche DISTINCT *nome-colonna* viene restituito il numero di righe che hanno soddisfatto la ricerca senza che il valore contenuto nella colonna specificata sia ripetuto.
MIN(*nome-colonna*)	Restituisce il valore minimo della colonna specificata tra le righe che soddisfano la ricerca.
MAX(*nome-colonna*)	Restituisce il valore massimo della colonna specificata tra le righe che soddisfano la ricerca.
SUM(*nome-colonna*)	Restituisce la somma dei valori della colonna specificata tra le righe che soddisfano la ricerca. La colonna specificata deve essere di tipo numerico.
AVG(*nome-colonna*)	Restituisce la media dei valori della colonna specificata tra le righe che soddisfano la ricerca. La colonna specificata deve essere di tipo numerico.

☞ *Rispetto allo standard SQL, picoSQL non permette di usare una funzione aggregata in un'espressione aritmetica.*

5.2 Funzioni numeriche

Nella descrizione di queste funzioni quando viene riportata la parola *espr-num* (o *espr-num-1*, *espr-num-2* etc.) si intende un'espressione numerica.

ABS(*espr-num*)	Restituisce il valore assoluto dell'espressione data.
ACOS(*espr-num*)	Restituisce l'arcocoseno dell'espressione data.
ASIN(*espr-num*)	Restituisce l'arcoseno dell'espressione data.
ATAN(*espr-num*)	Restituisce l'arcotangente dell'espressione data.
CEILING(*espr-num*)	Restituisce il minimo intero superiore o uguale all'espressione data.
COS(*espr-num*)	Restituisce il coseno dell'espressione data.
COT(*espr-num*)	Restituisce la cotangente dell'espressione data.
EXP(*espr-num*)	Restituisce il numero *e*, la base dei logaritmi naturali, elevata alla potenza definita dall'espressione data.
FLOOR(*espr-num*)	Restituisce il massimo intero inferiore o uguale all'espressione data.
LOG(*espr-num*)	Restituisce il logaritmo naturale dell'espressione data.
LOG10(*espr-num*)	Restituisce il logaritmo in base 10 dell'espressione data.
ROUND(*espr-num-1*, *espr-num-2*)	Restituisce il valore di *espr-num-1* arrotondato al numero di decimali specificato in *espr-num-2*.
SIGN(*espr-num*)	Restituisce -1 se l'espressione data vale un numero negativo, 1 se l'espressione data vale un numero positivo e 0 se l'espressione vale 0.
SIN(*espr-num*)	Restituisce il seno dell'espressione data.
SQRT(*espr-num*)	Restituisce la radice quadrata dell'espressione data.
TAN(*espr-num*)	Restituisce la tangente dell'espressione data.
TRUNCATE(*espr-num-1, espr-num-2*)	Restituisce il valore di *espr-num-1* troncato al numero di decimali specificato in *espr-num-2*.

5.3)Funzioni stringa

Nella descrizione di queste funzioni quando viene riportata la parola *stringa* (o *stringa-1, stringa-2* etc.) si intende un'espressione stringa.

ASCII(*stringa*)	Restituisce un valore numerico corrispondente al valore ASCII del primo carattere di *stringa*.
CHAR(*espr-num)*	Restituisce una stringa di un carattere il cui valore ASCII corrisponde al valore di *espr-num*.
CONCAT(*stringa-1, stringa-2*)	Restituisce una stringa formata dalla concatenazione delle due stringhe fornite come argomento.
CONVERT(*espr-num*,SQL_CHAR)	Restituisce una stringa contenente la rappresentazione del numero espresso da *espr-num.*
CONVERT(*stringa*,SQL_DOUBLE)	Restituisce un numero corrispondente al valore rappresentato in *stringa*. Se il valore contenuto in *stringa* non può essere trasformato in un numero, allora questa funzione restituisce 0.

CURRENT_TIMESTAMP(*stringa*)

Restituisce una stringa contenente il timestamp corrispondente al momento in cui è stato eseguito il comando, formattato secondo il contenuto di *stringa*. La formattazione prevede l'uso dei seguenti caratteri speciali:

`Y` cifra dell'anno;
`M` cifra del mese;
`D` cifra del giorno;
`H` cifra dell'ora;
`N` cifra del minuto;
`S` cifra del secondo;
`T` cifra del millesimo di secondo (sempre 0).

I caratteri diversi da questi, vengono riportati letteralmente. Per ottenere il formato standard del timestamp è necessario dunque fornire la seguente stringa formato:

```
'YYYY-MM-DD HH:NN:SS.TTT'
```

LCASE(s*tringa*)
LOWER(s*tringa*)

Restituisce una stringa come quella passata come argomento ma formata da tutte lettere minuscole. LCASE e LOWER sono sinonimi.

LENGTH(s*tringa*)

Restituisce la lunghezza della stringa fornita come argomento.

LOCATE(s*tringa-1, stringa-2, espr-num*)

Cerca la prima occorrenza di *stringa-1* all'interno di *stringa-2* iniziando dal carattere nella posizione specificata in *espr-num*. Se viene individuata una tale occorrenza, la funzione restituisce un numero che indica la posizione del primo carattere di tale occorrenza, altrimenti restituisce 0. Le posizioni dei caratteri sono calcolate a partire da 1.

LTRIM(*stringa*)

Restituisce una stringa corrispondente a quella passata come argomento ma priva dei caratteri spazio presenti prima del primo carattere non spazio.

RTRIM(*stringa*)

Restituisce una stringa corrispondente a quella passata come argomento ma priva dei caratteri spazio presenti dopo l'ultimo carattere non spazio.

SPACE(*espr-num*)

Restituisce una stringa formata da tanti caratteri spazio quanti ne sono specificati nell'argomento.

SUBSTRING(s*tringa*,
espr-num-1, *espr-num-2*)

Restituisce la sottostringa della stringa data come argomento partendo dalla posizione specificata da *espr-num-1* e lunga tanti caratteri quanti specificati in *espr-num-2*. La posizione dei caratteri sono calcolate partendo da 1.

UCASE(s*tringa*)
UPPER(s*tringa*)

Restituisce una stringa come quella passata come argomento ma formata da tutte lettere maiuscole. UCASE e UPPER sono sinonimi.

5.4)Funzioni temporali

Queste funzioni debbono avere come argomento un oggetto di tipo DATE, TIME o TIMESTAMP. Argomenti stringa non sono accettati mentre sono valide le sequenze di escape per date, tempi e date con il tempo.

DAYOFMONTH(*timestamp*)	Restituisce un numero, compreso tra 1 e 31, corrispondente al giorno del mese presente nell'argomento.
HOUR(*timestamp*)	Restituisce un numero, compreso tra 0 e 23, corrispondente all'ora presente nell'argomento.
MILLISECOND(*timestamp*)	Restituisce un numero, compreso tra 0 e 999, corrispondente ai millisecondi presenti nell'argomento.
MINUTE(*timestamp*)	Restituisce un numero, compreso tra 0 e 59, corrispondente ai minuti presenti nell'argomento.
MONTH(*timestamp*)	Restituisce un numero, compreso tra 1 e 12, corrispondente al mese presente nell'argomento.
NOW()	Restituisce un oggetto di tipo timestamp con la data e l'ora corrente.
SECOND(*timestamp*)	Restituisce un numero, compreso tra 0 e 59, corrispondente ai secondi presenti nell'argomento.
YEAR(*timestamp*)	Restituisce un numero, compreso tra 1 e 9999, corrispondente all'anno presente nell'argomento.

6) Condizioni di ricerca

Le condizioni di ricerca sono formate da uno o più predicati, combinati tra loro tramite gli operatori logici AND, OR e NOT. Esse sono usate per scegliere un sottoinsieme di righe da una tabella o da un insieme di più tabelle.
I predicati di una condizione di ricerca vengono valutati ogni volta che viene letta una riga dal database e se il risultato della valutazione risulta soddisfatto allora la riga viene restituita come valida, altrimenti viene ignorata.

☞ *SQL standard usa una logica a tre stati, VERO, FALSO e SCONOSCIUTO dove quest'ultimo valore viene assegnato ogni volta si esegue un confronto dove compare un valore NULL. Le tabelle di verità di questo tipo di logica per gli operatori logici menzionati sono le seguenti:*

AND	*VERO*	*FALSO*	*SCON.*
VERO	*VERO*	*FALSO*	*SCON.*
FALSO	*FALSO*	*FALSO*	*FALSO*
SCON.	*SCON.*	*FALSO*	*SCON.*

OR	*VERO*	*FALSO*	*SCON.*
VERO	*VERO*	*VERO*	*VERO*
FALSO	*VERO*	*FALSO*	*SCON.*
SCON.	*VERO*	*SCON.*	*SCON.*

NOT	
VERO	*FALSO*
FALSO	*VERO*
SCON.	*SCON.*

picoSQL usa una più semplice logica booleana a due valori, VERO e FALSO. Questo comporta un'unica differenza di comportamento tra lo standard SQL e picoSQL e cioè che il primo non ritiene soddisfatto un predicato che controlla se due valori NULL sono uguali mentre il secondo si. PicoSQL permette in particolare di fare join su campi a NULL, aggirando in tal modo la necessità delle outer-join, che comunque sono implementate.

Due operandi confrontati in un predicato, direttamente o indirettamente, debbono essere di tipi comparabili.

6.1)Predicato di confronto

Un predicato di confronto è formato da due espressioni confrontate da un'operatore, nella forma seguente:

espressione-1 operatore-di-confronto espressione-2

Gli operatori di confronto ammessi sono i seguenti:

=	uguale a
<> (oppure !=)	diverso da
>	maggiore di
>=	maggiore o uguale a
<	minore di
<=	minore o uguale a

6.2)Predicato BETWEEN

Un predicato BETWEEN controlla se un valore è contenuto tra due valori e ha la forma:

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

Questo predicato, senza il NOT, è equivalente a:

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

Il NOT nega il risultato del confronto in modo analogo a quanto fa l'operatore logico omonimo.

6.3)Predicato IN

Il predicato IN confronta il valore di un'espressione con una serie di valori e ha la forma:

espressione [NOT] IN (*valore-1*[,*valore-2*] ...)

Questo predicato, senza il NOT è equivalente a:

espressione = *valore-1* [OR *espressione* = *valore-2*] ...

Il NOT nega il risultato del confronto in modo analogo a quanto fa l'operatore logico omonimo.

6.4)Predicato LIKE

Il predicato LIKE consente di effettuare delle ricerche su stringhe utilizzando dei caratteri jolly. La forma del predicato è la seguente:

espressione-stringa [NOT] LIKE *stringa-ricerca*

All'interno della stringa di ricerca, i caratteri sono interpretati nel modo seguente:

- il carattere di sottolineatura (_) indica un qualsiasi carattere;
- il carattere di percentuale (%) indica una qualsiasi sequenza di 0 o più caratteri;
- il carattere barra rovesciata (\) indica che il carattere successivo deve essere considerato letteralmente anche se è un carattere jolly;
- ogni altro carattere viene considerato letteralmente.

Per esempio la stringa di ricerca `'B%'` restituisce vero per ogni valore di colonna che inizia con la lettera B, mentre la stringa di ricerca `'B__'` restituisce vero per ogni valore di colonna che inizia con la lettera B ed è lungo 3 caratteri.
Il NOT nega il risultato del confronto in modo analogo a quanto fa l'operatore logico omonimo.

6.5)Predicato NULL

Questo predicato permette di controllare se un'espressione è uguale a NULL e ha la forma:

espressione IS [NOT] NULL

Il NOT nega il risultato del confronto in modo analogo a quanto fa l'operatore logico omonimo.
Questo in SQL standard è l'unico modo per controllare l'esistenza di un valore NULL in una colonna. In picoSQL confrontando due valori di colonna uguali a NULL si ottiene il valore VERO.

6.6)Predicato EXISTS

Questo predicato permette di controllare l'esistenza di almeno una riga in una qualsiasi tabella che soddisfi determinate condizioni e ha la forma:

EXISTS (*sotto-interrogazione*)

Il risultato del predicato è VERO se *sotto-interrogazione*, che è una interrogazione indipendente ottenuta tramite il comando SELECT, restituisce almeno una riga, FALSO altrimenti.

7) Elenco dei comandi

Questo capitolo riporta in ordine alfabetico tutti i comandi SQL supportati nella release attuale da picoSQL.

ALTER TABLE

Sintassi: Formato 1:

ALTER TABLE *nome-tabella* RENAME *nuovo-nome-tabella*

Formato 2:

ALTER TABLE *nome-tabella* ADD [COLUMN] *nome-colonna definizione-dato*

Formato 3:

ALTER TABLE *nome-tabella* DROP [COLUMN] *nome-colonna*

Scopo: Permette modificare una tabella modificandone il nome oppure aggiungendo una colonna o infine eliminando una colonna.

Vedi anche: CREATE TABLE, DROP INDEX

Descrizione: Questo statement modifica la struttura di una tabella esistente senza alterarne il contenuto.

Il formato 1 consente di modificare il nome di una tabella.

Il formato 2 consente di aggiungere una nuova colonna a una tabella, che viene posta in fondo alle colonne esistenti. Se la tabella contiene già delle righe, la nuova colonna conterrà tutti valori NULL. Il nome della nuova colonna deve essere univoco all'interno dei nomi di colonna della tabella. La clausola COLUMN è solo documentativa e non altera il comportamento del comando.

Il formato 3 permette di eliminare una colonna da una tabella. La colonna da eliminare può contenere dati, e in tal caso questi vengono perduti, ma non può fare parte di un indice: in questo caso è necessario prima cancellare l'indice e quindi rimuovere la colonna.

CALL

Sintassi: CALL *nome-procedura* ([*argomento*[, *argomento*]...])

Scopo: Permette di invocare una stored procedure.

Vedi anche: SELECT

Descrizione: Questo statement permette di eseguire una stored procedure precedentemente creata. Il numero di argomenti può essere fisso o variabile in dipendenza di come viene dichiarata la procedura. Gli argomenti sono espressioni analoghe a quelle che è possibile specificare nell SELECT.
Come qualsiasi altro comando SQL, una CALL restituisce solo se ha avuto successo o meno; essa può generare un set di risultati, in modo analogo al comando SELECT, oppure non generare alcunché, come un comando INSERT, UPDATE o DELETE. Introdotta a partire dalla release 2.0.

CREATE INDEX

Sintassi: CREATE [UNIQUE] INDEX *nome-indice*
ON *nome-tabella*
(*nome-colonna* [ASC|DESC] [, *nome-colonna* [ASC|DESC]] ...)

Scopo: Permette la creazione di un indice su una tabella specificata. Gli indici sono importanti per migliorare le performance sul database.

Vedi anche: DROP

Descrizione: Questo statement crea un indice sulle colonne specificate della tabella specificata. Gli indici sono utilizzati automaticamente da picoSQL per migliorare le performance del database per quanto riguarda la ricerca di righe e per l'ordinamento delle stesse in presenza della clausola ORDER BY sia crescenti che decrescenti.

Un indice può essere creato su una tabella in qualsiasi momento, tanto se contiene dei dati quanto se è vuota.

La clausola UNIQUE assicura che nella tabella specificata non esistano due righe con valori uguali nelle colonne specificate come indice.

Il *nome-indice* in picoSQL non viene usato, ma deve essere fornito per compatibilità con lo standard SQL. Agli indici viene assegnato un nome convenzionale formato nel modo seguente:

nome-tabella_numero-progressivo

Il numero progressivo parte da 02 perché il numero 01 è riservato alla chiave primaria. Il nome convenzionale degli indici deve essere usato nella DROP INDEX per eliminare un indice esistente. Non rimangono mai buchi nei numeri progressivi, per cui se si cancella l'ennesimo indice, quelli successivi assumono un nuovo nome con il numero progressivo diminuito di 1.

Un indice può essere formato al massimo da 8 parti, dove con parte si intende un insieme di colonne adiacenti di tipo non nativo (CHAR, VARCHAR, NUMERIC, DATE, TIMESTAMP), e la sua occupazione può essere al massimo di 255 caratteri.

La clausola DESC, che permette di creare un indice con ordinamento decrescente, non è supportata nella release attuale.

CREATE TABLE

Sintassi: CREATE TABLE *nome-tabella*
(*nome-colonna definizione-dato* [PRIMARY KEY]
[, *nome-colonna definizione-dato* [PRIMARY KEY]] ...)
[, PRIMARY KEY (*nome-colonna* [,*nome-colonna*]...)]

Scopo: Permette la creazione di una tabella sul database.

Vedi anche: DROP

Descrizione: Questo statement crea una nuova tabella sul database, formata dalle colonne specificate.

La clausola PRIMARY KEY specificata dopo la dichiarazione di una colonna fa in modo che questa faccia parte di un indice univoco creato sulla tabella. A differenza dello standard SQL, questa clausola può apparire in più definizioni di colonne, creando in tal modo un indice primario formato da più colonne.
La clausola PRIMARY KEY specificata dopo la dichiarazione di tutte le colonne permette di specificare un indice primario su un numero qualsiasi di colonne in un qualsiasi ordine. Non è permesso dichiarare entrambi i tipi di clausole.

CREATE VIEW

Sintassi: CREATE VIEW *nome-vista* [(*nome-colonna* [, *nome-colonna*] ...)]
AS *interrogazione-semplice* [WITH CHECK OPTION]

Scopo: Permette la creazione di una vista sul database.

Vedi anche: DROP, SELECT

Descrizione: Questo statement crea una nuova vista sul database. Una vista serve per dare una visione particolare del database anche se i dati sono memorizzati in modo diverso. In pratica si tratta di una interrogazione, fatta usando lo statement SELECT senza le clausole ORDER BY e GROUP BY né alcuna funzione aggregata, che viene memorizzata e che può venire usata per nuove interrogazioni come una qualsiasi tabella. Le viste però non esistono fisicamente sul database e vengono derivate ciascuna volta che vengono usate.

Una vista può ereditare i nomi delle colonne dall'interrogazione che la crea oppure è possibile definirne di nuovi all'atto della creazione; in quest'ultimo caso il numero dei nomi specificati deve corrispondere a quello delle colonne restituite dall'interrogazione.

Quando si crea una vista che mette in join più tabelle, picoSQL richiede che la condizione di join sia espressa con una *clausola-join* (vedi il comando SELECT).

Le viste che riguardano una sola tabella possono essere aggiornate come una qualsiasi tabella. Se viene specificata la clausola WITH CHECK OPTION, ogni volta che si modifica una riga della vista, viene eseguita una verifica che la nuova riga o la modifica sulla riga esistente sia congruente con le condizioni specificate nella clausola WHERE della SELECT usata per creare la vista stessa. Se la verifica fallisce, la tabella non viene aggiornata e viene restituito un errore.

Sintassi: DELETE FROM *nome-tabella*
[WHERE {*condizione-di-ricerca*|CURRENT OF *nome-cursore*}]

Scopo: Permette l'eliminazione di un numero qualsiasi di righe da una tabella.

Vedi anche: INSERT, UPDATE, SELECT

Descrizione: Questo statement elimina dalla tabella specificata tutte le righe che soddisfano la condizione di ricerca. Se la clausola WHERE non viene specificata, vengono cancellate tutte le righe della tabella. La clausola CURRENT OF può essere usata al posto della condizione di ricerca per cancellare solo la riga correntemente letta da un cursore. Questa clausola può però essere usata solo da un linguaggio di programmazione in quanto è necessario usare una API per sapere il nome di un cursore (SQLGetCursorName) o per assegnarlo (SQLSetCursorName).

Sintassi: DROP { INDEX *nome-indice* | TABLE *nome-tabella* | VIEW *nome-view* }

Scopo: Permette l'eliminazione di un indice, di una tabella o di una vista.

Vedi anche: CREATE INDEX, CREATE TABLE, CREATE VIEW

Descrizione: Questo statement elimina definitivamente dal database un oggetto, che può essere un indice, una tabella o una vista.
Nel caso di DROP INDEX lo spazio ricuperato rimane allocato e viene riusato per gli indici rimasti o per la creazione di nuovi indici.
Nel caso di DROP TABLE, lo spazio su disco occupato dalla tabella con i relativi indici viene immediatamente rilasciato e reso disponibile per qualsiasi applicazione.
Nel caso di DROP VIEW lo spazio rilasciato, che è sempre molto limitato perché corrisponde solo alla lunghezza del comando usato nella creazione della vista, viene riusato nella creazione di nuove viste.

Sintassi: Formato 1:

INSERT INTO *nome-tabella* [*(nome-colonna* [, *nome-colonna*]...)]
VALUES (*espressione* |NULL [, *espressione* | NULL] ...)

Formato 2:

INSERT INTO *nome-tabella* [*(nome-colonna* [, *nome-colonna*]...)]
interrogazione

Scopo: Permette l'inserimento di una o più righe in una tabella

Vedi anche: REPLACE, DELETE, SELECT

Descrizione: Questo comando è usato per aggiungere una o più righe a una tabella di database.
Il formato 1 permette l'inserimento di una singola riga con i valori specificati dopo la clausola VALUES. Se viene fornita la lista opzionale di attributi, i valori dopo la clausola VALUES finiscono in essi con una corrispondenza posizionale (il primo valore nel primo attributo, il secondo nel secondo e così via). Se la lista degli attributi non è completa, negli attributi mancanti viene inserito il valore NULL. Se invece la lista opzionale di attributi non viene specificata, dopo la clausola VALUES devono essere specificati tanti valori quanti attributi conta la tabella e anche in questo caso viene fatta una corrispondenza posizionale. La riga viene inserita nel database in una posizione arbitraria.

Il formato 2 permette di prelevare i valori da inserire da una qualsiasi tabella tramite un'apposita operazione SELECT. Anche in questo caso la corrispondenza tra attributi specificati, o della tabella se non viene specificata la lista opzionale degli attributi, e quelli restituiti dalla SELECT viene fatta posizionalmente.

Se uno o più attributi nella tabella sono stati dichiarati di tipo SERIAL e contengono il valore NULL, a essi viene assegnato un numero sequenziale. Il valori assegnati possono essere ricuperati con il comando SELECT SERIAL (disponibile dalla release 2.0) a patto che esso venga eseguito sulla stessa statement handle usata per la INSERT.

REPLACE

Sintassi: Formato 1:

REPLACE INTO *nome-tabella* [(*nome-colonna* [, *nome-colonna*]...)]
VALUES (*espressione* |NULL [, *espressione* | NULL] ...)

Formato 2:

REPLACE INTO *nome-tabella* [(*nome-colonna* [, *nome-colonna*]...)]
interrogazione

Scopo: Permette l'inserimento o la sovrascrittura di una o più righe in una tabella

Vedi anche: INSERT, DELETE

Descrizione: Questo comando è analogo a INSERT, con la differenza che nel caso la riga non possa essere inserita a causa di un indice univoco che verrebbe duplicato, REPLACE sovrascrive l'intera riga che contiene il valore univoco. Da un punto di vista logico quindi usare REPLACE equivale ad eseguire prima un DELETE sulla tabella in base al primo indice univoco e successivamente una INSERT con i valori specificati. In pratica però l'operazione viene fatta in modo atomico, vale a dire che sulla tabella non esiste un momento in cui siano assenti sia la vecchia riga che la nuova, ed è molto più efficiente dell'esecuzione di DELETE e INSERT. Un errore per indice duplicato quindi si può avere solo se sulla tabella sono presenti più di un indice univoco e se la sovrascrittura provocasse la duplicazione di un indice univoco successivo al primo.

Sintassi: Formato 1:

SELECT [ALL | DISTINCT] *lista-di-espressioni*
FROM { *nome-tabella* [[AS] *alias*][, *nome-tabella* [[AS] *alias*]] ... |
clausola-join }
[WHERE *condizione-di-ricerca*]
[GROUP BY *nome-colonna* [, *nome-colonna*] ...]
[HAVING *condizione-ricerca*]
[UNION *comando-select*]
[ORDER BY {*numero-intero* [ASC|DESC]|*espressione* [ASC | DESC]}
[, {*numero-intero* [ASC|DESC]|*espressione* [ASC | DESC]}]...]
[LIMIT *numero-intero*]
[OFFSET *numero-intero*]

Formato 2:

SELECT *lista-di-espressioni*
FROM *nome-tabella*
[WHERE *condizione-di-ricerca*]
FOR UPDATE

Formato 3 (release >= 2.0):

SELECT SERIAL

Scopo: Permette di interrogare il database e di ottenerne informazioni sulla base di condizioni logiche di ricerca.

Vedi anche: CREATE VIEW, UNION, INSERT

Descrizione: Questo comando è il più complesso di tutto il linguaggio poiché fornisce i mezzi necessari per interrogare il database e individuare un insieme di dati.
Per l'uso del formato 3, vedere INSERT.
Il formato 2 è un sottoinsieme del formato 1, con in più solo la clausola FOR UPDATE, e viene usato nei programmi per fare in modo che la riga che si sta leggendo non sia modificabile da un altro utente. La riga torna disponibile quando si legge la riga successiva o quando viene chiuso il cursore.
Vediamo quindi tutte le clausole del formato 1.

ALL | DISTINCT
Se viene specificata la clausola DISTINCT non vengono visualizzate le righe duplicate. Questa è chiamata 'proiezione' del risultato del comando. Va comunque notato che per poter eliminare le righe duplicate, picoSQL esegue un ordinamento dei risultati, il che può portare a un incremento dei tempi di risposta, specialmente se il numero di righe selezionate è molto ampio.
La clausola ALL viceversa lascia tutte le righe e rappresenta il comportamento di default quando non viene specificato alcunché.

lista-di-espressioni
La *lista-di-espressioni* indica quali sono i dati da visualizzare ogni volta che viene trovata una riga con le caratteristiche richieste; la forma di *lista-di-espressioni* è la seguente:

{ * | *espressione* [[AS] *alias*] [, *espressione* [[AS] *alias*]]... }

Se viene specificato un asterisco (*), esso rappresenta tutte le colonne di tutte le tabelle che vengono menzionate nella successiva clausola FROM. È permesso usare funzioni aggreganti come espressioni.
Alle espressioni possono essere assegnati dei nomi *alias*. Tali nomi vengono visualizzati normalmente in testa alle colonne dei risultati nei programmi d'interrogazione interattivi, ma possono servire anche per distinguere, nelle clausole successive come ORDER BY e GROUP BY, colonne con nomi uguali ma appartenenti a tabelle diverse.

FROM
Permette di specificare la lista delle tabelle da cui si vuole estrarre i dati. Nel caso in cui l'estrazione avvenga da più di una tabella, normalmente è necessario indicare una o più condizioni di join, altrimenti si ottiene come risultato la combinazione di ciascuna riga di ciascuna tabella con tutte le altre (prodotto cartesiano). Una join è una condizione che lega una colonna di una tabella a una condizione di un'altra tabella. In picoSQL si possono specificare join in due modi:

- mettendo la condizione insieme a tutte le altre nella WHERE (vedi seguito);
- specificando le condizioni direttamente nella FROM con una sintassi opportuna;

Usando il primo modo, la clausola FROM si riduce a una semplice lista di nomi di tabella. Si possono specificare dei nomi *alias* delle tabelle in modo da rendere possibili le 'self-join', cioè join di una tabella con se stessa.

Il secondo modo è apparentemente più complesso ma ha il vantaggio di separare le condizioni di join dalle condizioni di ricerca e di poter impostare le join esterne (outer join). La sintassi della *clausola-join* è la seguente:

nome-tabella [[AS] *alias*] {INNER | LEFT OUTER| RIGHT OUTER}
JOIN {*nome-tabella* [[AS] *alias*] | *clausola-join*}
ON *condizione-di-join*

Come si vede è possibile impostare un numero qualsiasi di join, anche di tipo diverso tra loro, annidando più *clausole-join*.
Mettere la condizione di join nella WHERE o usare la clausola INNER JOIN fa ottenere gli stessi risultati ma può influenzare le performance: nel primo caso infatti è picoSQL che sceglie l'ordine ottimale di scansione delle tabelle in modo da minimizzare le letture mentre nel secondo caso l'ordine di scansione è fissato nell'ordine con cui sono riportate le tabelle. Questo può risultare svantaggioso quando si debbano fare delle query dinamiche nelle quali non si conosce a priori su quale colonna di quale tabella verrà fatta la ricerca.

WHERE
Permette di specificare una *condizione-di-ricerca* che limita il numero delle righe ottenute. Anche le condizioni di join possono essere messe dopo questa clausola. Vedi anche "Condizioni di ricerca".

GROUP BY
Permette di raggruppare più righe dal risultato della ricerca sul database. Quando si usa questa clausola, in *lista-di-espressioni* possono essere messe solo funzioni aggreganti o nomi di colonna riportate anche in GROUP BY. Il risultato ottenuto contiene una riga per ciascun valore distinto delle colonne elencate, che viene definita 'gruppo'. Eventuali funzioni aggreganti specificate vengono applicate a ciascun gruppo.

HAVING
Questa clausola può essere specificata solo dopo una GROUP BY e serve per limitare il numero di gruppi selezionati. Essa infatti permette di definire una *condizione-di-ricerca* nella quale possono comparire anche le funzioni aggreganti definite nella *lista-di-espressioni*.

ORDER BY
Ordina il risultato dell'interrogazione. Ciascun elemento nella lista specificata dopo ORDER BY può avere la clausola ASC, che indica che è richiesto l'ordinamento ascendente, oppure la clausola DESC che indica che è richiesto l'ordinamento discendente. Nel caso che nessuna di queste due clausole venga specificata, viene sottinteso l'ordinamento ascendente. Se la lista contiene un numero intero N, l'ordinamento viene fatto sull'elemento N-esimo della *lista-di-espressioni*.

LIMIT
Limita il numero di righe restituite al numero specificato come argomento. Se il numero totale di righe selezionate è inferiore al numero specificato come argomento, la clausola non ha alcun effetto.

OFFSET
Restituisce le righe selezionate saltandone un numero pari al numero specificato come argomento. Se il numero totale di righe selezionate è inferiore al numero specificato come argomento, restituisce un messaggio di dati non trovati.

Sintassi: *select-senza-order-by* UNION [ALL] *select-senza-order-by*
[UNION [ALL] *select-senza-order-by*] ...
[ORDER BY *intero* [ASC | DESC][, *intero* [ASC | DESC]] ...]

Scopo: Permette di fare l'unione del risultato di due o più comandi SELECT

Vedi anche: SELECT

Descrizione: La UNION è un'operazione che permette di combinare il risultato di due o più comandi SELECT, opzionalmente ordinati per le colonne specificate. Ciascuna SELECT deve restituire un uguale numero di colonne, che debbono corrispondere anche come tipo, e non può specificare una propria ORDER BY. La UNION normalmente elimina le colonne duplicate a meno di non specificare la clausola ALL.
La clausola ORDER BY è analoga a quella usata nel comando SELECT con la limitazione che si possono specificare solo numeri interi corrispondenti alle posizioni delle colonne da ordinare.

UPDATE

Sintassi: UPDATE *nome-tabella*
SET *nome-colonna* = {*espressione* | NULL}
[, *nome-colonna* = {*espressione* | NULL}] ...
[WHERE {*condizione-di-ricerca*|CURRENT OF *nome-cursore*}]

Scopo: Permette di modificare una o più righe sul database

Vedi anche: INSERT, DELETE, SELECT

Descrizione: Questo comando è usato per modificare una o più righe su una tabella. Ciascuna colonna prende il valore dell'espressione o NULL corrispondente. Nell'espressione può comparire anche il nome della colonna che si sta modificando, che avrà il valore appena letto.
La clausola WHERE permette di specificare l'insieme di righe su cui viene effettuato l'aggiornamento: nel caso tale clausola non venga specificata, vengono aggiornate tutte le righe della tabella. La clausola CURRENT OF può essere usata al posto della condizione di ricerca per modificare solo la riga correntemente letta da un cursore. Questa clausola può però essere usata solo da un linguaggio di programmazione in quanto è necessario usare una API per sapere il nome di un cursore (SQLGetCursorName) o per assegnarlo (SQLSetCursorName).

8) Ottimizzatore delle query

picoSQL ha al proprio interno un ottimizzatore che trova il modo migliore per soddisfare le interrogazioni richieste. Sono previste solo due strategie di lettura, la scansione sequenziale di un'intera tabella oppure l'utilizzo degli indici dichiarati.
picoSQL si ricava la possibilità di usare un indice analizzando la condizione di ricerca ed è per questo motivo che è importante, specialmente su tabelle con molte righe, la definizione degli indici e l'utilizzo di condizioni di ricerca che li sfruttino.
Per illustrare in modo semplice come deve essere strutturata una condizione, supponiamo intanto di avere un indice di una sola colonna di nome I.

L'ottimizzatore di picoSQL ricava la propria strategia di ricerca analizzando l'interrogazione in modo *formale*, non considerando cioè i valori effettivi con cui le colonne vengono confrontate né, di conseguenza, i valori realmente presenti nella base dati. Per capire questo punto, bisogna considerare che spesso vengono fatte delle query usando i parametri, come nel caso seguente:

```
I = ?
```

Questa query può essere eseguita più volte, sostituendo ogni volta un valore diverso al posto del punto interrogativo. È evidente che in casi come questo il modo più efficiente di procedere è di eseguire un'unica analisi del comando e stabilire una strategia di ricerca valida per qualsiasi parametro.
Fare l'analisi conoscendo il valore del parametro costringerebbe a ripetere l'analisi ogni volta che quest'ultimo viene modificato, non apportando, nella quasi totalità dei casi, informazioni utili a migliorare la strategia di ricerca.
Su questi presupposti, picoSQL riesce sempre a individuare un indice di ricerca, purché ne esista uno usabile.

Affinché un indice possa essere usato per una ricerca, una condizione deve individuare sulla colonna dell'indice uno o più *intervalli esaustivi* di valori che contengano tutte le righe richieste, come, per esempio, la seguente:

```
I < 5
```

Per il fatto che l'ottimizzazione non tiene conto dei dati può accadere che l'utilizzo di un indice non porti alcun vantaggio: supponendo, per esempio, che nel database siano presenti solo righe in cui la colonna I ha sempre un valore inferiore a 5, la condizione di ricerca precedente non limita in alcun modo la lettura della tabella e sarebbe stata più efficiente una scansione sequenziale.

Condizioni in AND a una condizione che individua un intervallo esaustivo non influenzano la valenza di un indice, per cui anche la seguente condizione permette l'utilizzo dell'indice:

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

Le cose cambiano se, al posto di una AND si usa una OR. Nella condizione seguente:

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

non è detto che le righe che ci interessano rientrino nell'intervallo definito dall'indice (I < 5) per cui tale intervallo non è più esaustivo e l'uso dell'indice diventa inutile. La cosa non migliora neppure se ci fosse un'indice definito sulla colonna A, in quanto neanche questo intervallo sarebbe esaustivo.

L'analisi di entrambi gli intervalli, con eliminazione di eventuali sovrapposizioni, sarebbe pesante da un punto di vista di calcolo e potrebbe portare anche a una doppia scansione completa della tabella. In questi casi quindi viene eseguita una scansione sequenziale.
Quest'ultima osservazione ci induce a una prima semplice regola empirica: nelle ricerche è meglio non usare l'operatore logico OR perché porta a scandire l'intera tabella. Questo però non è sempre vero: ci sono infatti casi in cui anche in presenza di OR si riesce a individuare degli intervalli esaustivi, come, per esempio:

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

In questo caso gli intervalli sono due, ma racchiudono tutte le righe richieste. picoSQL è in grado di sfruttare gli indici in presenza di più intervalli esaustivi, anche nel caso che tali intervalli si sovrappongono, come nel caso:

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

In presenza di indici composti da più colonne, la teoria non cambia, anche se può essere più complicato capire se la ricerca impostata può essere fatta usando un indice o meno.
Per prima cosa l'elemento in posizione *n* di un indice composto può essere usato in una ricerca se è *n* = 1 oppure se può essere usato l'elemento in posizione *n - 1*. Supponiamo di avere un indice composto formato dalle colonne I, J e K e di avere le seguenti condizioni di ricerca:

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

Nella prima ricerca è possibile usare l'indice in tutte le sue parti, nella seconda si può usare solo la prima parte dell'indice mentre nella terza l'indice non può essere usato.
Fatta questa considerazione, le regole di utilizzo di un indice sono le stesse già viste per indici semplici. Analizzando le seguenti condizioni:

```
(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
```

si può notare che la prima usa completamente tutti e tre gli indici mentre la seconda utilizza solo la componente sulla colonna I; in tal caso infatti J e K possono avere un valore qualsiasi e quindi non possono essere usati come indice. In realtà, analizzando i valori attentamente si nota che sarebbe possibile individuare un intervallo esaustivo più stretto, ma questo comporterebbe un analisi contestuale che picoSQL non fa.