SQL

18 maggio 2023

Indice

Parte I

Linguaggio

Capitolo 1

Introduzione

SQL (Structured Query Language) è:

- un linguaggio relazionale per basi di dati: permette di definire la struttura dei database, memorizzare, interrogare e proteggere i dati;
- è un linguaggio *non procedurale*: gli statement del linguaggio permettono di specificare quali dati si devono reperire e non come devono essere trovati;
- è *case insensitive*; ogni istruzione termina con un punto e virgola. Può esser buona norma scrivere le keyword SQL in maiuscolo, per distinguerle da identificatori di tabelle ecc di una data applicazione.

1.1 Backus-Naur form

Per descrivere la sintassi dell'SQL si farà uso della notazione Backus Naur Form (BNF), un formalismo di metasimboli che si aggiungono alla sintassi SQL pura per descrivere compattamente le possibilità espressive offerte dal linguaggio.
I metasimboli aggiuntivi propri della notazione BNF sono:

```
<>   ::=   |   []   ...   {}   ;   "
```

Nello specifico:

- le `<>` isolano/identificano un termine della sintassi;
- la `::=` è usato per definire un concetto/statemente;
- le `[]` che il termine all'interno è opzionale;
- la `|` indica che deve esser scelto uno tra i termini separati dalle barre
- le `{}` che i simboli al suo interno formano un gruppo (ad esempio utilizzato con `|` per delimitare le scelte);
- le `()` invece sono parte integrante del linguaggio SQL.
- i `...` indicano che qualcosa dopo il quale sono posti può essere ripetuto 1 o n volte; combinarli con `[ ]...` permette di indicare che un certo elemento può apparire nessuna, una o n volte;

- il ; serve per evitare la ripetizione di definizione di oggetti defiti identicamente; ad esempio

  ```
  <character literal> ;
  <varchar literal> ;
  <long varchar literal> ::= <character string>
  ```

  è equivalente a

  ```
  <character literal> ::= <character string>
  <varchar literal> ::= <character string>
  <long varchar literal> ::= <character string>
  ```

1.2 Costanti ed espressioni

1.2.1 Costanti

Una *costante* (*literal*) rappresenta un valore immutabile, utilizzato in espressioni e comandi SQL. La BNF delle costanti SQL è presentata in seguito

```
<literal> ::=
   <numeric literal>      |
   <alphanumeric literal> |
   <temporal literal>     |
   <boolean literal>      |
   <hexadecimal literal>

<numeric literal> ::=
   <integer literal>      |
   <decimal literal>      |
   <float literal>

<integer literal> ::= [+|-] <whole number>

<decimal literal> ::=
   [+|-] <whole number> [ .<whole number>] |
   [+|-] <whole number>.                   |
   [+|-] .<whole number>

<float literal> ::=
   <mantissa> {E|e} <exponent>

<alphanumeric literal> ::= <character list>

<temporal literal> ::=
   <date literal>      |
   <time literal>      |
   <timestamp literal>

<date literal> ::= ' <years> - <months> - <days>'
```

```
<time literal> ::= ' <hours> : <minutes> [: <seconds>]'

<timestamp literal> ::=
  ' <years> - <months> - <days> <space>  <hours> : <minutes> [: <seconds>]'

<boolean literal> ::= TRUE | FALSE

<hexadecimal literal> ::= X <character list>

<years> ::= <whole number>

<months>  ;
<days>    ;
<hours>   ;
<minutes> ;
<seconds> ::= <digit> [<digit>]

<mantissa> ::= <decimal literal>
<exponent> ::= <integer literal>

<character list> ::= ' [<character> ...] '
<character> ::= <digit> | <letter> | <special char> | ''

<whole number> ::= <digit>...
```

Ogni costante è caratterizzata da un *tipo di dato*, che descrive i valori rappresentabili.

1.2.2 Espressioni

Una espressione consiste in una o più operazioni, possibilmente tra parentesi, rappresentanti un valore di un determinato tipo di dato.
La BNF:

```
<expression> ::=
    <scalar expression>  |
    <row expression>     |
    <table expression>
```

Ponendo enfasi sulla complessità del valore restituito possiamo classificare le espressioni in:

- *scalar expression*: espressione che restituisce un singolo valore (es `4`);
- *row expression*: espressione che restituisce una tupla.

  ```
  (1, 'John', '1983-12-12', 12*13)
  ```

- *table expression*: espressione che restituisce una tabella.

```
(  (1, 'John', '1983-12-12')  ,
   (2, 'Jack,  '1981-01-02')  ,
   (3, 'Al,    '1985-03-04')  )
```

Ad esempio una `select` è una *table expression*, poiché restituisce una serie di tuple.

1.2.2.1 Scalar expression

Considerando il numero di componenti dell'espressione possiamo distinguere in forma *semplice* (*singular*) o *composta* (*compound*):

```
<scalar expression> ::=
    <singular scalar expression> |
    <compound scalar expression>

<singular scalar expression> ::=
    <literal>               |
    <column specification>  |
    <user variable>         |
    <system variable>       |
    <cast expression>       |
    <case expression>       |
    NULL                    |
    ( <scalar expression> ) |
    <scalar function>       |
    <aggregation function>  |
    <scalar subquery>
```

Singular scalar expression Una singular scalar expression può contenere, tra le altre:

- una *costante*;
- una *specificazione di colonna*: identifica una specifica colonna di una tabella; definito come

  ```
  <column specification> ::=
      [[ <database_name> .] < table_name > . ] < column name >  [[AS] <Alias>]
  ```

  il valore prodotto è il contenuto della colonna;
- una `case expression`: tipo un if-then else per ricodificare compattamente, avente definizione formale

  ```
  <case expression> ::=
     CASE <when definition> [ ELSE <scalar expression> ] END

  <when definition> ::= <when definition-1> | <when definition-2>

  <when definition-1> ::=
  ```

Funzione	significato
`ABS`	valore assoluto
`CONCAT`	fa il paste di stringhe
`CURRENT_DATE`	restituisce la data odierna
`CURRENT_TIME`	restituisce ora
`CURRENT_TIMESTAMP`	restituisce data ora
`DATABASE`	mostra il nome del database corrente
`DATE`	coercisce una stringa a data
`DATEDIFF`	coercisce una stringa a data
`DAYOFMONTH`	estrae il giorno del mese da una data
`DEFAULT`	restituisce il valore di default di una data colonna
`EXP`	esponenziale base numero di nepero
`LN`	logaritmo naturale
`IF`	tipo l'ifelse di R
`ISNULL`	testa se il valore è `NULL`
`LENGTH`	lunghezza di una stringa
`MONTH`	estrae mese di una data
`RAND`	ritorna un umero casuale tra 0 e 1
`REPLACE`	rimpiazza testo in una stringa
`ROUND`	arrotonda
`SUBSTRING`	estrae pezzi da una stringa

Tabella 1.1: Funzioni scalari notevoli

Funzione	significato
`AVG`	media
`COUNT`	numero di valori in una colonna (o in una tabella)
`MIN`	minimo
`MAX`	massimo
`SUM`	somma dei valori in una colonna
`STDDEV`	deviazione standard

Tabella 1.2: Funzioni di aggregazione notevoli

```
  <scalar expression>
  WHEN <scalar expression> THEN <scalar expression>
  [ WHEN <scalar expression> THEN <scalar expression> ]...

<when definition-2> ::=
  WHEN <condition> THEN <scalar expression>
  [ WHEN <condition> THEN <scalar expression> ]...
```

Un esempio:

```
SELECT PLAYERNO, JOINED, TOWN,
       CASE
         WHEN JOINED >= 1980 AND JOINED <= 1982
            THEN 'Seniors'
         WHEN TOWN = 'Eltham'
            THEN 'Elthammers'
         WHEN PLAYERNO < 10
            THEN 'First members'
         ELSE 'Rest' END
FROM PLAYERS
```

- una `cast expression` serve per convertire un tipo di dato in un altro; la sintassi può essere evinta dal seguente esempio

  ```
  CAST('1997-01-15' AS DATE)
  ```

- `NULL`

- una *espressione tra parentesi*: ogni espressione può esser ugualmente tra un numero di parentesi a piacimento, e ciò non cambia il valore della stessa

- *funzione scalare*: una funzione (vedi manuale DBMS per le funzioni messe a disposizione dal sistema, alcune comuni sono riportate in tab 1.1). Le funzioni scalari possono anche esser *nested* (ad esempio `ROUND(SQRT(2),2)`

- una *funzione di aggregazione*: differentemente da una funzione scalare opera su più tuple e le sommarizza (alcune funzioni comuni sono riportate in tab 1.2);

- una *subquery scalare*: in generale una *subquery* è una `<table expression>` posta tra parentesi

  ```
  <subquery> ::= ( <table expression> )
  ```

  impiegata all'interno di un'altra table expression.
  L'utilizzo di una subquery all'interno della table expression madre, è legato alla tipologia della subquery. In SQL si distingue, a seconda del risultato restituito, tra quattro tipi di subquery:

  1. *table* subquery: restituiscono una tabella
  2. *row* subquery: restituiscono una riga

3. *column* subquery: restituiscono una colonna
4. *scalar* subquery: restituiscono uno scalare

Per ciò che limitatamente riguarda le scalar subquery queste posson esser poste ovunque possa essere posta una costante.
Vedremo l'utilizzo delle subquery (table, column) anche in specifiche parti delle interrogazioni (`from`, `where`).

Compound scalar expression In una `<compound scalar expression>` compaiono uno o più operatori (es algebrici, relazionali ecc), che legano fra loro una o più espressioni semplici.

1.2.2.2 Row expression

Le row expression vengono tipicamente utilizzate nello statement `INSERT`: rappresentano una riga con almeno un valore. Nei sistemi che lo ammettono, una row subquery può essere sostituita al posto di una `<row expression>`.
La BNF:

```
<row expression> ::=
    <singular row expression>

<singular row expression> ::=
   ( <scalar expression> [ { ,<scalar expression> } ... ] )  |
   <row subquery>
```

Le *compound row expression* non sono ancora supportate in SQL. Ad esempio non esiste `(1,2,3)+(4,5,6)`..

1.2.2.3 Table expression

Infine le table expression: questa rappresenta un insieme di righe; ad esempio in una `insert` può essere utilizzata per insirire molteplici righe contemporaneamente.
La BNF introdotta verrà ripresa nella sezione delle interrogazioni SQL.

1.3 SELECT statement, table expressions e subquery

A partire da questa sezione approfondiamo lo statement per effettuare interrogazioni sulla base dati (ipotizzando che questa sia già stata creata).

1.3.1 SELECT statement

La sintassi di un `SELECT` statement coincide quasi con una `table expression`, nel senso che la sua definizione BNF:

```
<select statement> ::=
  <table expression>
  [ for clause ]
```

anche se per i nostri fini le possiamo considerare equivalenti [1].

1.3.2 Table expression

Una table expression ha una definizione BNF:

```
<table expression> ::=
  { <select block head>         |
    <subquery>                  |
    <compound table expression> }
  [ <select block tail> ]

<select block head> ::=
  <select clause>
  [ <from clause>
  [ <where clause> ]
  [ <group by clause> ]
  [ <having clause> ] ]

<subquery>::= ( <table expression> )

<compound table expression> ::=
  <table expression> <set operator> <table expression>

<set operator> ::=
   {UNION | INTERSECT | EXCEPT} [ALL]

<select block tail> ::=
  <order by clause>
```

Una table expression può essere utilizzata all'interno non solo di un `SELECT` statement ma in vari punti della sintassi SQL; alcuni punti rilevanti:

- ogni table expression (e ogni `SELECT` statement) constano almeno di una *select clause*; le altre clause (`WHERE`, `GROUP BY`, `ORDER BY`) come sono opzionali;
- se clause come `WHERE`, `GROUP BY` e/o `HAVING` sono impiegate è necessaria `FROM`;
- l'ordine con cui si specificano le varie clause è prefissato; ad esempio una `GROUP BY` non può mai venire prima di una `WHERE` o `FROM`, mentre la `ORDER BY` se presente è sempre l'ultima;
- `HAVING` (non??) può essere usata in assenza di `GROUP BY`.

Ipotizzando che tutte siano state specificate, l'interpretazione delle varie clausole della table expression avviene nel seguente ordine:

1. si parte da `from`, mediante la quale si selezionano i dati in ingresso;

[1] La `for clause` viene sviluppata nel capitolo 26 del van der lans "Introduction to Embedded SQL"

2. si prosegue con `where` che seleziona le righe che proseguono mediante una condizione valutata riga per riga
3. si continua con `group by` che raggruppa righe sulla base di ugual valori nelle colonne specificate ...
4. ed `having` che eventualmente seleziona i gruppi che rispettano una condizione;
5. si ordinano i record ottenuti mediante `order by`;
6. si selezionano le colonne desiderate mediante `select`

Nel seguito si introdurranno le varie clausole.

1.3.3 Subquery e compound table expression

Una table expression può avere tre forme:

- la prima è quella "standard" (detta *select block*) che abbiamo visto sin'ora;
- la seconda è quella della subquery, che altro non è altro che una table expression inclusa tra parentesi. Le parentesi sono spesso utili quando diversi *select block* avvengono entro la medesima table expression
- la terza è la combinazione di table expression attraverso operatori insiemistici

1.4 SELECT statement: la clausola FROM

Se una table expression contiene una `FROM`, l'elaborazione parte da questa; l'effetto della `FROM` è di scegliere i dati.
La BNF della clausola è:

```
<from clause> ::=
  FROM <table reference> [, { <table reference> }]...

<table reference> ::=
  { <table specification> | <table subquery> | <join specification> }
  [[AS] <pseudonym> ]

<table specification> ::=
  [<database name> . |<user name> .] <table name> [ [AS] <pseudonym> ]

<join specification> ::=
  <table reference> <join type> <table reference> [ <join condition> ]

<join condition> ::=
  ON <condition> | USING <column list>

<join type> ::=
  [ INNER ] JOIN        |
```

```
  LEFT  [ OUTER ] JOIN  |
  RIGHT [ OUTER ] JOIN  |
  FULL  [ OUTER ] JOIN  |
  UNION JOIN            |
  CROSS JOIN

<column list> ::=
  ( <column name> [ , <column name> ]... )

<table subquery> ::=
     ( <table expression> )

<column specification> ::=
  [ <table specification> . ] <column name>
```

La scelta della/e tabelle da utilizzare è effettuata meidante una table reference, che può consistere alternativamente in una table specification, una table subquery o un join, che andiamo ad approfondire nel prosieguo.
Nel caso in cui nella `FROM` si specifichino più `<table reference>` (separate da virgola), il server restituisce il **prodotto cartesiano** tra le tabelle specificate[2].

1.4.1 Table specification

Nella `<table specification>`:

- per `<table name>` si intede il nome di una tabella o di una vista;
- a precedere una table name può esservi, a seconda del prodotto sql, il nome di un utente proprietario della tabella o il nome del database[3] di cui la tabella fa parte.

Column specification Nel prosieguo ci si può riferire a `column specification`, per indicare una specifica colonna: per indicarla basta aggiungere un punto e il nome della colonna alla table specification (qualora questa sia necessaria).

1.4.2 Table subquery

Una subquery, ovvero una table expression tra parentesi, può esser utilizzata all'interno di una `from`: l'effetto è che la tabella sulla quale verranno effettuati i processi a seguire viene generata on the fly a partire dalla selezione impostata. Ad esempio per ottenere il numero di giocatori capitani di una squadra che gioca nella prima divisione:

```
SELECT SMALL_TEAMS.PLAYERNO
FROM   (SELECT PLAYERNO, DIVISION
        FROM TEAMS) AS SMALL_TEAMS
WHERE  SMALL_TEAMS.DIVISION = 'first'
```

[2] Un primo modo di fare dei join è specificare due tabelle in `from` e selezionare i record uguali sugli attributi chiave delle due tabelle con `where`

[3] In questo secondo caso spesso e volentieri (dipende dal prodotto) è possibile evitare di specificare il nome del database, se in precedenza si è usato si è impostato il database di default (`USE` con mysql, o da linea di comando per postgres).

Si noti che possono esser impiegate solo *table* subquery; non *scalar*, *row* o *column*.

1.4.3 Join

Sebbene un join possa esser specificato facendo utilizzo del prodotto cartesiano delle due tabelle specificate in `from` e della selezione impiegabile in `where` (*join implicito*), il modo migliore per farlo è direttamente esplicitamente.
Riprendendo il chunk di BNF:

```
<join specification> ::=
  <table reference> <join type> <table reference> [ <join condition> ]

<join condition> ::=
  ON <condition> | USING <column list>

<join type> ::=
  [ INNER ] JOIN        |
  LEFT  [ OUTER ] JOIN  |
  RIGHT [ OUTER ] JOIN  |
  FULL  [ OUTER ] JOIN  |
  UNION JOIN            |
  CROSS JOIN

<column list> ::=
  ( <column name> [ , <column name> ]... )
```

`join type` specifica quale join usare, e ad esso si possono sostituire:

- `INNER` per il join interno, il tradizionale theta-join dell'algebra relazionale. È l'opzione di default, quindi la keyword `inner` è omettibile;
- `RIGHT OUTER`, `LEFT OUTER` o `FULL OUTER` per il join esterno (`OUTER` è comunque omettibile);
- `CROSS` e `UNION` li possiamo tranquillamente ignorare.

Le colonne che formano il legame tra le due tabelle sono specificate nella `join condition` che assume due forme:

- se si utilizza `ON`, `<condition>` è una espressione che confronta i campi delle relazioni che determinano il join. Tipicamente il confronto è una semplice uguaglianza, ma espressioni più complesse (con connettivi logici) sono ammissibili[4]. Ad esempio matricola e voto dell'esame sostenuto dalle studentesse

  ```
  select studente.matricola, esame.voto
  from   esame INNER JOIN studente
         ON studente.matricola=esame.studente
  where  sesso='F'
  ```

[4] Van Der Lans, da pag 202

- `USING` può essere impiegata, come forma contratta, quando le due tabelle hanno chiavi con i medesimi nomi e la condizione di selezione sarebbe l'uguaglianza (come spesso è): ad esempio il seguente chunk di query

  ```
  FROM TEAMS INNER JOIN PLAYERS
       USING (PLAYERNO)
  ```

  equivale a

  ```
  FROM TEAMS INNER JOIN PLAYERS
       ON TEAMS.PLAYERNO = PLAYERS.PLAYERNO
  ```

Come desumibile dalla sintassi, nella condizione `from` possono esser specificati più di un join. Es per trovare cognome dello studente, nome del corso e voto d'esame degli esami

```
SELECT cognome, voto, corso.nome
FROM   (esame INNER JOIN corso
        ON corso.codice = esame.corso)
            INNER JOIN studente
            ON esame.studente = studente.matricola
```

1.4.4 L'uso di pseudonimi

Gli `pseudonym` sono nomi alternativi, specificati dopo un opzionale `AS` e temporanei, e servono tipicamente per:

- abbreviare il riferimento ad una determinata tabella. Un esempio di merge artigianale:

  ```
  SELECT PAYMENTNO, PEN.PLAYERNO, AMOUNT,
         NAME, INITIALS
  FROM   PENALTIES AS PEN, PLAYERS AS P
  WHERE  PEN.PLAYERNO = P.PLAYERNO
  ```

- nella realizzazione di interrogazioni nidificate, come si vedrà nel seguito;

- per far riferimento più volte ad una stessa tabella in una medesima `from`; si può pensare allo pseudonimo in effetti come ad una singola copia della relazione (che contiene tutte le tuple dell'originale). In alcuni casi bisogna utilizzare due (o più) copie della stessa relazione, pertanto si dichiarano due pseudonimi per la stessa. Un esempio; per estrarre tutti gli impiegati che hanno lo stesso cognome (e diverso nome) di impiegati della Produzione:

  ```
  select I1.Cognome  , I1.Nome
  from   Impiegato I1, Impiegato I2
  where  I1.Cognome =  I2.Cognome   and
         I1.Nome    <> I2.Nome      and
         I2.Dipart  =  'Produzione'
  ```

L'altro modo di utilizzare lo pseudonimo è nella `<select clause>`, dove verrà impiegato per ridenominare l'apposita colonna nel risultato.

C2	C1	C1 and C2	C1 or C2	not C1
True	True	True	True	False
True	False	False	True	False
True	Unknown	Unknown	True	False
False	True	False	True	True
False	False	False	False	True
False	Unknown	False	Unknown	True
Unknown	True	Unknown	True	Unknown
Unknown	False	False	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown

Tabella 1.3: Tabella di verità operatori booleani di SQL

1.5 SELECT statement: la clausola WHERE

La clausola `WHERE` ammette come argomento una `condition`, ovvero una espressione booleana che viene valutata per ciascuna riga, ed è costruita combinando (eventualmente) *predicati* con gli operatori boleani e parentesi. La BNF è:

```
<where clause> ::= WHERE <condition>

<condition> ::=
  <predicate>                 |
  <predicate> OR <predicate>  |
  <predicate> AND <predicate> |
  ( <condition> )             |
  NOT <condition>

<predicate> ::=
   <predicate with comparison> |
   <predicate with in>         |
   <predicate with beetween>   |
   <predicate with like>       |
   <predicate with null>       |
   <predicate with exists>     |
   <predicate with any all>
```

Un predicato/espressione può restituire i valori *vero*, *falso* o *unknown* (questo quando uno qualsiasi dei termini del predicato ha valore nullo); `where` seleziona solo le righe che valutano l'espressione vera. Per invertire i vari predicati apporvi davanti un `NOT`.
La tabella di verità per le varie condizioni è riportata in tabella 1.3

1.5.1 Predicati con confronti

In questo predicato due espressioni sono confrontate per via di operatori di confronto. La BNF è:

```
<predicate with comparison> ::=
    <scalar expression> <comparison operator> <scalar expression> |
    <row expression>    <comparison operator> <row expression>
```

```
<comparison operator> ::=
    = | <> | < | > | <= | >=
```

dove gli operatori di comparazione sono i consueti, ad eccezione di `<>` che indica "diverso da" (l'equivalente di `!=` del C).
Gli operatori di cui sopra forniscono un ordinamento anche per le stringhe (ordinmento ASCII).

Comparazione con subquery Una `scalar expression` può essere sostituita da una subquery che ritorni uno scalare; per questo possiamo implementare anche delle query con delle subquery in `WHERE`.
Ad esempio per ottenere numero, nome iniziali di ogni giocatore più anziano del giocatore con numero 8467:

```
SELECT PLAYERNO, NAME, INITIALS
FROM PLAYERS
WHERE BIRTH_DATE <
      (SELECT BIRTH_DATE
       FROM   PLAYERS
       WHERE LEAGUENO = '8467')
```

1.5.2 Subquery correlate

Quando in una subquery ci riferiamo ad una colonna della riga che stiamo considerando nella main query, si dice che la subquery è correlata.
Ad esempio per ottenere il numero di match giocati da giocatori che vivono a inglewood:

```
SELECT MATCHNO
FROM   MATCHES
WHERE 'Inglewood' =
      (SELECT TOWN
       FROM   PLAYERS
       WHERE  PLAYERS.PLAYERNO = MATCHES.PLAYERNO)
```

Si nota che abbiamo sfruttato `MATCHES.PLAYERNO` che è un singolo valore (nella `where` processiamo una riga alla volta) della main query. Allora la subquer `SELECT TOWN ...` si dice correlata.
Le subquery correlate non possono essere eseguite in una volta sola, ad esempio prima di eseguire la main query; debbono eseguite di volta in volta, riga per riga, durante l'esecuzione della main query (poiché dipendono da opportuni valori di questa).[5]

1.5.3 Predicati con `IN`

Il predicato con `in` serve per verificare se il valore dell'espressione a sinistra dell'operatore `in` è contenuta nella lista a destra dello stesso.
Ha due per due forme, determinate :

[5] Per approfondire i concetti di scope delle subquery ed altri esempi vedere ulteriori esempi cfr van der lans pag 303-314.

```
<predicate with in> ::=
   <scalar expression> [NOT] IN <scalar expression list> |
   <scalar expression> [NOT] IN <column subquery>        |
   <row expression>    [NOT] IN <row expression list>    |
   <row expression>    [NOT] IN <table subquery>

<row expression list> ::=
   ( <scalar expression list>
     [ {, <scalar expression list> }...] )

<scalar expression list> ::=
    ( <scalar expression>  [ {, <scalar expression > }...] )

<column subquery> ;
<table subquery>::= (<table expression>)
```

La differenza principale dal predicato con la expression list alla subquery è che la seconda è determinabile on the fly, mentre la prima deve essere prespecificata (magari i valori da scrivere sono pure numerosi).
Chiaramente `NOT` inverte il valore dell'espressione booleana.
Ad esempio per trovare numero e nome dei dipendenti nati nel 1939, 1940 o 1963:

```
SELECT NUMERO,NOME
FROM DIPENDENTI
WHERE YEAR(NATO) IN (1939, 1940, 1963);
```

Per ottener il match e il numero di set vinti e persi di tutti i match dove si sono vinti o persi due set:

```
SELECT MATCHNO, WON, LOST
FROM   MATCHES
WHERE  2 IN (WON, LOST)
```

Per un esempio dell'uso delle subquery il numero giocatore e nmome di ogni giocatore che ga giocato almeno un match per il primo team:

```
SELECT PLAYERNO, NAME
FROM   PLAYERS
WHERE  PLAYERNO IN
       (SELECT PLAYERNO
        FROM   MATCHES
        WHERE  TEAMNO = 1)
```

1.5.4 Predicati con `BEETWEEN`

`beetween` serve per determinare se un determinato valore rientra in un dato range:

```
<predicate with beetween> ::=
     <scalar expression> [NOT] BEETWEEN
         <scalar expression> AND <scalar expression>
```

Ad esempio:

```
SELECT PLAYERNO, BIRTH_DATE
FROM   PLAYERS
WHERE  BIRTH_DATE BETWEEN '1962-01-01' AND '1964-12-31'
```

1.5.5 Predicati con `LIKE`

SQL mette a disposizione l'operatore `LIKE` per il confronto delle stringhe;

```
<predicate with like> ::=
     <scalar expression> [NOT] LIKE <like pattern>
     [ESCAPE <character>]

<like pattern> ::= <scalar alphanumeric expression>
```

dove `pattern` ammette come caratteri speciali:

- `_`: matcha un carattere arbitrario;
- `%`: matcha da 0 a n caratteri arbitrari

Ad esempio per estrarre gli impiegati con un cognome con "o" in seconda posizione e terminanti per "i"

```
select *
from   Impiegato
where  Cognome like '_o%i'
```

Infine `ESCAPE` serve per specificare una lettera di escape che venga utilizzata se si desidera cercare proprio i due caratteri speciali nel pattern:

```
select ..
where NAME LIKE '%#_%' ESCAPE '#'
```

In questo modo riusciamo a cercare anche `_` nel pattern, escapeandolo con `#`.

1.5.6 Predicati con `IS NULL`

Per selezionare i termini con valori nulli, SQL fornisce il predicato `IS NULL`:

```
<predicate with null> ::=
    <scalar expression> IS [NOT] NULL
```

Questo fa sì che il predicato risulti vero solo se l'attributo ha valore nullo; con l'aggiunta di `NOT` si ha la negazione.
Ad esempio, per dare il nome dei fornitori di cui non si conosce la via:

```
SELECT NOME
FROM FORNITORI
WHERE VIA IS NULL
```

1.5.7 Predicati con EXISTS

La BNF:

```
<predicate with exists> ::= EXISTS <table subquery>

<table subquery> ::= (<table expression>)
```

restituisce il valore vero solo se `table subquery` contiene almeno una tupla.
Per un esempio nel seguito si seleziona il nome dei giocatori che hanno avuto almeno una penalità:

```
SELECT NAME
FROM   PLAYERS
WHERE  EXISTS
       (SELECT *
        FROM PENALTIES
        WHERE PLAYERNO = PLAYERS.PLAYERNO)
```

Per ogni giocatore della tabella `PLAYER` SQL determina se la subquery ritorna una riga; in altre parole conrolla se vi è vi è un risultato non vuoto e manda avanti la riga solo in tale evenienza.
Di fatto è irrilevante ciò che si specifica nel `SELECT` statement della subquery, dato che sono solamente `FROM` e `WHERE` a determinare se la subquery ha delle tuple.
Ovviamente si può invertire la selezione apponendo `NOT` davanti ad `EXISTS` (la possibilità si evince dalla BNF di `WHERE`).

1.5.8 Predicati con ANY, ALL, SOME

Nel caso in cui una subquery restituisca un set di valori invece di uno solo (in questo caso basta un confronto singolo), il confronto ha significato solo se si indica un *quantificatore*.
La BNF:

```
<predicate with any all> ::=
    <scalar expression> <any all operator> <column subquery>

<any all operator> ::=
    <comparison operator> {ALL | ANY | SOME}

<column subquery> ::= ( <table expression> )
```

Se è specificato `ALL`:

- il predicato è *true* se la subquery non restituisce valori o il confronto è vero per tutti i valori restituiti;
- il predicato è *false* se il confronto è falso per almeno un valore restituito;
- il predicato è *unknown* se il confronto non è falso per alcun valore restituito e almeno un confronto è unknown a causa di valori `NULL`.

Se è specificato `SOME` o `ANY` (i due possono essere usati intercambiabilmente, il loro uso è identico), allora:

- il predicato è *true* se il confronto è vero per ciascun valore di almeno una riga restituita.
- il predicato è *false* se la subquery non restituisce valori o il confronto è falso per almeno un valore di ogni riga restituita.
- il predicato è *unknown* se il confronto non è vero per alcuna riga restituita e almeno un confronto è unknown a causa di valori `NULL`

Ad esempio, per dare nome e quantità dei prodotti presenti in quantità massima:

```
SELECT NOME,QUANTITA
FROM PRODOTTI
WHERE QUANTITA >= ALL
   (SELECT QUANTITA
    FROM PRODOTTI)
```

Per dare nome e quantità dei prodotti presenti in quantità non minima:

```
SELECT NOME,QUANTITA
FROM PRODOTTI
WHERE QUANTITA > ANY
  (SELECT QUANTITA
   FROM PRODOTTI)
```

1.6 SELECT statement: la clausola SELECT e le funzioni di aggregazione

```
<select clause> ::=
  SELECT [ DISTINCT | ALL ]  <select element list>

<select element list> ::=
  <select element> [ , <select element> ]... |
  *

<select element> ::=
  <scalar expression> [[ AS ] <column name> ] |
  <table specification>.*                     |
  <pseudonym>.*

<column name> ::= <name>
```

Nella clausola `SELECT` possono comparire alternativamente:

- l'asterisco `*`: rappresenta la selezione di tutti gli attributi della relazione risultante dagli step precedenti o, se successivo ad una `table specification`[6] (o `pseudonym`), l'insieme degli attributi provenienti da quella tabella.
  Ad esempio il seguente statement seleziona tutte le colonne post join provenienti dalla tabella `PENALTIES`

[6] Questo si fa tipicamente se nella `FROM` si sono selezionate più di una tabella

```
SELECT PENALTIES.*
FROM   PENALTIES INNER JOIN TEAMS
       ON PENALTIES.PLAYERNO = TEAMS.PLAYERNO

SELECT PEN.*
FROM   PENALTIES AS PEN INNER JOIN TEAMS
ON     PEN.PLAYERNO = TEAMS.PLAYERNO
```

- generiche espressioni (separate da virgole) sul valore degli attributi di ciascuna riga selezionata.
  Ad esempio per estrarre l'età da una tabella dove è presente l'anno di nascita:

  ```
  select cognome, YEAR(current_date())-YEAR(nascita) AS eta FROM studente;
  ```

  Le espressioni possono essere le più discrezionali possibili, ad esempio se per ogni match vogliamo il numero, la parola `Tally`, la differenza tra `WON` e `LOST` e il valore di `WON` moltiplicato per 10:

  ```
  SELECT  MATCHNO, 'Tally', WON - LOST, WON * 10
  FROM    MATCHES
  ```

  Per la costruzione di espressioni, SQL mette a disposizione un gran numero di funzioni.
  Infine le espressioni possono essere ridenominate, per motivi estetici o funzionali, mediante `AS`.

1.6.1 Rimozione delle righe duplicate con `DISTINCT`

L'eliminazione delle righe duplicate è ottenuta con la parola chiave `DISTINCT` da porre immediatamente dopo la parola chiave `SELECT`.
La ricerca dei duplicati avviene con riferimento solamente alle specificate nella `SELECT` clause, non a tutte le colonne derivanti dagli step precedenti.

L'utente potrebbe anche specificare la keyword `ALL`, allo stesso posto di `DISTINCT`: tuttavia di fatto è inutile poiché non cambia il comportamento di default (che è quello di restituire tutte le righe proveniente dagli step precedenti).

1.6.2 Funzioni di aggregazione

Le funzioni di aggregazione (anche dette funzioni colonna) possono essere applicate in una espressione della `SELECT` clause, e l'effetto è di elaborare un insieme di tuple riconducendole ad uno scalare.
Se la `table expression` che stiamo considerando:

- non presenta la clausola `GROUP BY`, la funzione di aggregazione opera su tutte le righe e restituisce una unica riga;

- alternativamente viene elaborata una riga per ogni gruppo specificato da `GROUP BY`.

1.6.2.1 Le funzioni disponibili

Lo standard SQL prevede le seguenti funzioni di aggregazione:

```
<aggregation function> ::=
COUNT    ( [ DISTINCT | ALL ] { * | <expression> } ) |
MIN      ( [ DISTINCT | ALL ] <expression> )         |
MAX      ( [ DISTINCT | ALL ] <expression> )         |
SUM      ( [ DISTINCT | ALL ] <expression> )         |
AVG      ( [ DISTINCT | ALL ] <expression> )         |
STDDEV   ( [ DISTINCT | ALL ] <expression> )         |
VARIANCE ( [ DISTINCT | ALL ] <expression> )
```

Anche qui si anticipa che `ALL` è inutile e può essere ignorata.

`COUNT` Questa funzione si comporta in maniera lievemente diversa dalle rimanenti

- se si specifica `*`, viene restituito il numero di righe del risultato. Ad esempio per contare il numero di giocatori:

  ```
  SELECT COUNT(*)
  FROM   PLAYERS
  ```

- se si specifica `expression`, si contano i record con `expression` non nulla. Ad esempio per contare il numero di league (disponibili)

  ```
  SELECT COUNT(LEAGUENO)
  FROM   PLAYERS
  ```

  Se si specifica `DISTINCT` le righe doppie su `expression` non vengono aggiunte al conteggio. Ad esempio per contare il numero di città:

  ```
  SELECT COUNT(DISTINCT TOWN)
  FROM   PLAYERS
  ```

`MIN`, `MAX`, `SUM`, `AVG`, `STDDEV` **e** `VARIANCE` Le rimanenti funzioni presentano una sintassi e un funzionamento simile:

- si calcola la funzione di aggregazione sulla `expression`;
- se specificato `DISTINCT` le tuple con `expression` vengono rese uniche prima di applicare la funzione di aggregazione.

1.6.2.2 Requirements di applicazione

Vi sono alcuni requirements generali per l'applicazione:

- quando si utilizza una funzione di aggregazione l'output consiste sempre di (almeno) una riga;
- le funzioni di aggregazione non possono essere composte/nestate; ad esempio non è valido `COUNT(MAX(...))`

- se la SELECT clause contiene una funzione di aggregazione, allora i nomi di colonne presenti nellaa SELECT clause devono necessariamente comparire entro una funzione di aggregazione. Ad esempio non è corretta la scrittura

```
SELECT COUNT(*), PLAYERNO
FROM   PLAYERS
```

  poiché il conteggio si fa ad una riga mentre PLAYERNO ne estrarrebbe diverse.

1.7 SELECT statement: la clausola GROUP BY

Spesso sorge l'esigenza di applicare una funzione di aggregazione separatamente a sottoinsiemi di righe: SQL mette a disposizione la clausola GROUP BY.
La clausola ammette come argomento un insieme di espressioni (semplici colonne la maggior parte delle volte) e l'interrogazione raggrupperà le righe che possiedono gli stessi valori per questo insieme di attributi; i NULL fanno gruppo a sè.
La BNF completa:

```
<group by clause> ::=
  GROUP BY <group by specification list> [ WITH { ROLLUP | CUBE } ]

<group by specification list> ::=
  <group by specification> [ , <group by specification> ]...

<group by specification> ::=
  <group by expression>          |
  <grouping sets specification>  |
  <rollup specification>

<group by expression> ::= <scalar expression>

<grouping sets specification > ::=
  GROUPING SETS ( <grouping sets specification list > )

<grouping sets specification list> ::=
  <grouping set specification> [ , <grouping set specification> ]...

<grouping set specification> ::=
  <group by expression>  |
  <rollup specification> |
  ( < grouping sets specification list> )

<rollup specification> ::=
  ROLLUP ( <group by expression list> ) |
  CUBE   ( <group by expression list> ) |
  (  )
```

1.7.1 Regole generali della GROUP BY clause

Le seguenti regole si applicano a tutte le table expression con GROUP BY:

1. in molti prodotti SQL, qualsiasi colonna specificata nella SELECT clause deve trovarsi alternativamente all'interno di una funzione di aggregazione, nella lista delle variabili per le quali si fa grouping o in entrambe le posizioni. Pertanto per la maggior parte dei prodotti, lo statement

   ```
   SELECT TOWN, COUNT(*)
   FROM   PLAYERS
   GROUP BY SEX
   ```

   non è corretto perché TOWN non si trova all'interno di una funzione di aggregazione o costituisce variabile di grouping (che qui è SEX).
   Questo perché il risultato di una funzione di aggregazione deve essere sempre una singola linea per gruppo;

2. spesso l'espressione usata per effettuare il grouping viene anche riportata nella SELECT clause, sebbene ciò non sia strettamente obbligatorio;

3. una espressione usata per effettuare grouping può anche comparire nella SELECT clause in una espressione composta di complessità arbitraria; ad esempio se l'espressione PLAYERNO * 2 compare nella GROUP BY clause, le espressioni PLAYERNO * 2 , (PLAYERNO * 2) - 100 e MOD(PLAYERNO * 2, 3) - 100 sono assolutamente valide; al contrario non possono comparirvi espressioni come PLAYERNO, 2 * PLAYERNO, PLAYERNO * 100 e 8 * PLAYERNO * 2;

4. espressioni che occorrono più volte in una GROUP BY clause sono rese uniche: ad esempio GROUP BY TOWN, TOWN è convertito a GROUP BY TOWN

1.7.2 Esempi di base

Un esempio di grouping su una colonna: per ogni città trova il numero di giocatori:

```
SELECT TOWN, COUNT(*) as NGIOCATORI
FROM   PLAYERS
GROUP  BY TOWN
```

Un esempio di grouping su due colonne: per ogni dipartimento e fornitore, dare il numero del dipartimento, il nome del fornitore, il numero e prezzo medio dei prodotti forniti dal fornitore al dipartimento:

```
SELECT NUMERO, NOME, COUNT(*) AS "N.PRODOTTI",
       AVG(PREZZO) AS "PREZZO MEDIO"
FROM   FORNISCE, TRATTA
WHERE  FORNISCE.CODICE = TRATTA.CODICE
GROUP  BY NUMERO, NOME
```

Come detto si possono usare le espressioni più generali per effettuare raggruppamento. Ad esempio per ogni anno presente nella tabella PENALTIES, ottenere il numero di penalità pagate:

```
SELECT YEAR(PAYMENT_DATE), COUNT(*)
FROM   PENALTIES
GROUP BY YEAR(PAYMENT_DATE)
```

1.8 SELECT statement: la clausola HAVING

A volte è necessario selezionare non tanto singole righe (fatto mediante `WHERE`) ma interi insiemi che si formano formati mediante `GROUP BY`: SQL mette a disposizione la clausola `HAVING`. La BNF:

```
<having clause> ::= HAVING <condition>
```

Similmente a `WHERE` la `condition` è espressione booleana, ma nella quale si possono porre funzioni di aggregazione, a contrario di quella in `WHERE`.

1.8.1 Esempi semplici

Ad esempio per trovare il prezzo medio di ogni prodotto con prezzo medio maggiore o uguale a 3.00:

```
SELECT CODICE, AVG(PREZZO) AS "PREZZO MEDIO"
FROM FORNISCE
GROUP BY CODICE
HAVING AVG(PREZZO) >= 3.00
```

1.8.2 Requirements

I requirements sono simili a quelli visti in precedenza per la `SELECT` clause e `GROUP BY`: ogni colonna specificata nell'HAVING deve essere riportata entro una funzione di aggregazione o nella lista di colonne nominate in `GROUP BY`. Pertanto lo statement seguente è incorretto

```
SELECT TOWN, COUNT(*)
FROM   PLAYERS
GROUP  BY TOWN
HAVING BIRTH_DATE > '1970-01-01'
```

perché `BIRTH_DATE` è un dato a livello di singola tupla e non è raggruppato o summarizzato.

1.9 SELECT statement: la clausola ORDER BY

Può esser necessario ordinare i risultati di una table expression sulla base di un criterio; occorre utilizzare la clausola `ORDER BY`, avente BNF

```
<order by clause> ::=
  ORDER BY <sort specification> [ , <sort specification> ]...

<sort specification> ::=
  <column name>       [ <sort direction> ] |
  <scalar expression> [ <sort direction> ] |
```

```
  <sequence number>   [ <sort direction> ]

<sort direction> ::= ASC | DESC
```

con le seguenti peculiarità:

- la sequenza delle variabili rispetto alle quali viene effettuato l'ordinamento è quella specificata dopo `BY`;
- per ciascuna `sort specification`, se non viene specificata la sort direction si assume ascendente (per cui `ASC` è di fatto inutile).

1.9.1 Le varie `sort specification`

In merito ai vari tipi di `sort specification`:

- `column name` è semplicemente il nome di una colonna; ad esempio se per ogni ordine serve dare data e nome del cliente, in ordine discendente di data:

  ```
  SELECT   DATA,NOME
  FROM     ORDINI
  ORDER BY DATA DESC
  ```

- `scalar expression` è una espressione (che spesso e volentieri fa uso di colonne della tabella). Ad esempio per ordinare la query sulla base della prima lettera del cognome:

  ```
  SELECT NAME, INITIALS, PLAYERNO
  FROM   PLAYERS
  ORDER BY SUBSTR(NAME, 1, 1)
  ```

- `sequence number`: se utilizziamo un intero, questo viene interpretato come indice selettore di variabile specificata nella `SELECT` clause. Ad esempio

  ```
  SELECT PAYMENTNO, PLAYERNO
  FROM   PENALTIES
  ORDER BY 2
  ```

  è equivalente a

  ```
  SELECT PAYMENTNO, PLAYERNO
  FROM   PENALTIES
  ORDER BY PLAYERNO
  ```

1.9.2 Sorting di valori `NULL`

`NULL` è gestito diversamente per quanto riguarda il sorting in diversi prodotti SQL. Consultare il manuale del proprio sistema.

1.10 Combinare table expression

A volte è necessario comporre i risultati provenienti da due diverse table expression mediante operatori insiemistici: quello che si ottiene è una `compound table expression`. Riprendendo il chunk di BNF che definisce le `table expression`:

```
<table expression> ::=
  { <select block head>         |
    <subquery>                  |
    <compound table expression> }
  [ <select block tail> ]

<compound table expression> ::=
  <table expression> <set operator> <table expression>

<set operator> ::=
   {UNION | INTERSECT | EXCEPT} [ALL]

<select block tail> ::=
  <order by clause>
```

I requirements affinché si possano utilizzare gli operatori insiemistici su due table expression riguardano gli attributi delle due table expression:

- devono essere in numero uguale;
- devono avere domini compatibili compatibili; fa fede in questo la posizione assunta all'interno della `SELECT` clause.

Nello specifico non è necessario che gli schemi su cui vengono effettuate le operazioni insiemistiche siano identici; il risultato normalmente usa i nomi del primo operando.
Gli operatori applicabili per combinare insiemisticamente due table expression sono[7] `UNION` per l' unione, `INTERSECT` per intersezione ed `EXCEPT` per la differenza insiemistica.
Detti operatori insiemistici effettuano di default una eliminazione dei duplicati (e tutti i `NULL` sono considerati uguali); tale eliminazione è disattivabile specificando `ALL`.
Una `ORDER BY` clause può essere specificata dopo l'ultima table expression per effettuare l'ordinamento del risultato complessivo.
A volte per leggibilità le due select vengono poste tra parentesi.
Ad esempio per trovare i codici dei dipendenti non manager con salario eccedente 1850:

```
SELECT CODICE
FROM   DIPENDENTI
WHERE  SALARIO > 1850
EXCEPT
SELECT CODICE
FROM DIPARTIMENTI
```

[7] Si noti che ogni interrogazione che faccia uso degli operatori `INTERSECT` e `EXCEPT` può esser espressa utilizzando altri costrutti del linguaggio (interrogazioni nidificate). `UNION` invece aumenta l'espressività del linguaggio SQL.

Capitolo 2

Creazione di base di dati

Quando si passa all'SQL, generalmente come convenzione di linguaggio si parla di tabella, al posto di relazione, di riga al posto di tupla, e di colonna o campo al posto di attributo. Qui i differenti termini si utilizzeranno in maniera intercambiabile.

2.1 Creazione di database

Ogni tabella è memorizzata all'interno di un database. Per crearne uno

```
CREATE DATABASE nome;
```

Oltre a questo è possibile impostare varie opzioni (caratteristiche del db, permessi ecc), ma queste cambiano fortemente in ragione del sistema.

2.2 Creazione di tabelle

Per creare tabelle si usa lo statement `CREATE TABLE`, la cui sintassi di base è:

```
<create table statement> ::=
  CREATE [ TEMPORARY ] TABLE [ IF NOT EXISTS ]
    <table specification> <table schema>

<table specification> ::= [ <database name> . ] <table name>

<table schema> ::= ( <table element> [ , <table element> ]... )

<table element> ::=
  <column definition> |
  <table integrity constraint>

<column definition> ::=
  <column name> <data type> [ <null specification> ]
    [ <column integrity constraint> ] [ <column option>... ]

<data type> ::=
```

```
  <numeric data type> |
  <alphanumeric data type> |
  <temporal data type> |
  <bolean data type> |
  <blob data type>

<numeric data type> ::=
  <integer data type> |
  <decimal data type> |
  <float data type>

<integer data type> ::=
  SMALLINT  |
  INTEGER   |
  INT       |
  BIGINT

<decimal data type> ::=
  DECIMAL [ ( <precision> [, <scale> ]) ] |
  DEC     [ ( <precision> [, <scale> ]) ] |
  NUMERIC [ ( <precision> [, <scale> ]) ] |
  NUM     [ ( <precision> [, <scale> ]) ]

<float data type> ::=
  FLOAT [ ( <length> ) ] |
  REAL  |
  DOUBLE [ PRECISION ]

<alphanumeric data type> ::=
  CHAR        [ ( <length> ) ] |
  CHARACTER   [ ( <length> ) ] |
  VARCHAR       ( <length> )   |
  CHAR VARYING  ( <length> )   |
  CHARACTER VARYING ( <length> ) |
  LONG VARCHAR

<temporal data type> ::=
  DATE       |
  TIME       |
  TIMESTAMP

<boolean data type> ::= BOOLEAN

<blob data type> ::= BLOB

<precision> ;
<scale>     ;
<length>    ::= <whole number>

<null specification> ::=  NOT NULL
```

```
<column option> ::=
  DEFAULT <literal> |
  COMMENT <alphanumeric literal>
```

Ogni tabella viene definita associandole un nome ed elencando gli attributi che ne compongono lo schema. Per ogni attributo si definiscono un nome, un dominio ed eventualmente un insieme di vincoli che devono esser rispettati dai valori dell'attributo. Dopo aver definito gli attributi si possono definire i vincoli che coinvolgono più attributi della tabella.

2.2.1 Tipi di dati

SQL mette a disposizione alcune famiglie di *domini* (tipi di dati) elementari da associare agli attributi delle tabelle/relazioni; è anche possibile definire nuovi domini-tipi di dati. In seguito le sintassi proposte sono non quelle più complete ed esaustive (si veda Van Der Lans a pg 490 per una guida SQL completa).

Interi Per memorizzare interi si possono utilizzare:

- `smallint` (rappresenta da -2^{15} a $2^{15}-1$)
- `integer`, abbreviabile a `int`, (rappresenta da -2^{31} a $2^{31}-1$)

Numeri virgola fissa In smallint e integer non è ammessa una rappresentazione della parte frazionaria e non si controlla la dimensione della rappresentazione.
`numeric`, avente sintassi

```
numeric [ (Precisione [, Scala] )]
```

serve per rappresentare reali con una parte decimale di lunghezza prefissata (es euro senza utilizzare dei costosi `float`). `Precisione` specifica il numero di cifre memorizzabili, con `Scala` quelle dopo la virgola. Es con `numeric(6,4)` si possono rappresentare valori compresi tra $-99,9999$ e $99,9999$.

Reali Vengono implementati mediante rappresentazione in virgola mobile, in cui a ciascun numero corrisponde una coppia di valori: la mantissa (valore frazionario) e l'esponente (intero). Il valore approssimato del reali si ottiene moltiplicando la mantissa per la potenza di 10 con grado pari all'esponente. Per esempio `0.17E16` è $1,7 \cdot 10^{15}$.
I tipi per i reali si suddividono a seconda che si desideri specificare la precisione della mantissa (ovvero il numero di cifre dedicate alla rappresentazione della mantissa). Il tipo che permette di specificarla (eventualmente) è

```
float[(Precisione)]
```

Alternativamente sono disponibili `real` e `double precision`, in cui la precisione della mantissa è data, ma dipende dall'implementazione (in ogni caso `double` ha precisione doppia rispetto a `real`).

Caratteri Per rappresentare caratteri-stringhe si usa `character`. Una sintassi semplificata è

```
character [varying] [(Lunghezza)]
```

La lunghezza delle stringhe di caratteri può esser fissa o variabile.Se `Lunghezza` non è specificata si intende 1. Per le stringhe di lunghezza variabile `Lunghezza` indica la lunghezza massima memorizzata in inserimento. SQL ammette anche le forme compatte, e molto utilizzate, `char` e `varchar`, rispettivamente per `character` e `character varying`

Istanti temporali Per rappresentarli si può utilizzare `date` per date, `time` per ore, e `timestamp` per data+ora

Dati binari Per rappresentarli si utilizza il tipo `blob` (introdotto con SQL-3, quindi non implementato ovunque).

2.2.2 NOT NULL

Il vincolo `not null` indica che il valore nullo non è ammesso come valore dell'attributo; in tal caso l'attributo deve esser sempre specificato (in fase di inserimento/modifica).
Se il sistema rileva una violazione del vincolo (si tenta di inserire un null), il comando di aggiornamento viene rifiutato, segnalando errore all'utente.

2.2.3 Column options

2.2.3.1 DEFAULT

E' possibile specificare il valore che deve assumere l'attributo in assenza di una sua specifica in scrittura (inserimento/modifica).
La sintassi per la specifica dei valori di default è:

```
default  < valore | user | null >
```

dove:

- `valore` è un valore coerente con il dominio
- `null` è il valore nullo
- `user` impone come valore l'identificativo dell'utente che esegue il comando di aggiornamento della tabella

Quando il valore di default non è specificato, il sistema usa un valore predefinito dipendente dal tipo di dato della colonna (solitamente il valore nullo).

2.2.3.2 COMMENT

Mediante `COMMENT` è possibile inserire un commeto sulla singola variabile.

2.2.4 Specifica di vincoli di integrità

`CREATE TABLE` permette la specifica dei vincoli di integrità intra/interrelazionali.
La BNF

```
<column integrity constraint> ::=
  PRIMARY KEY |
  UNIQUE      |
  <check integrity constraint>

<table integrity constraint> ::=
 [ CONSTRAINT  <constraint name> ]
 { <primary key>   |
   <alternate key> |
   <foreign key>   |
   <check integrity constraint> }

<primary key> ::= PRIMARY KEY  <column list>

<alternate key> ::= UNIQUE <column list>

<foreign key> ::= FOREIGN KEY  <column list> <referencing specification>

<referencing specification> ::=
  REFERENCES <table specification> [ <column list> ]
    [ <referencing action>... ]

<referencing action> ::=
  ON { UPDATE | DELETE } { CASCADE | RESTRICT | SET NULL | NO ACTION }

<column list> ::=  ( <column name> [ , <column name> ]... )

<check integrity constraint> ::= CHECK ( <condition> )
```

2.2.4.1 Column integrity constraint: `UNIQUE`

`unique` si applica ad un attributo o a un insieme di attributi di una tabella e impone che i valori dell'attributo (o le ennuple di valori sull'insieme di attributi) costituiscano una chiave, cioè righe differenti della tabella non possano avere gli stessi valori. Viene fata eccezione per il valore nullo (si assume che i valori nulli siano tutti diversi tra loro).
La definizione può avvenire mediante due sintassi a seconda che l'`unique` sia definito su uno o più attributi:

- quando bisogna definire il vincolo su un solo attributo si fa seguire la specifica dell'attributo da `unique`

  ```
  Matricola char(6) unique
  ```

- nel caso si imponga unicità su più attributi, nella creazione della tabella dopo averne definito gli attributi, si usa la sintassi

```
unique( Attributo {, Attributo} )
```

Un esempio è dato da:

```
Nome     varchar(6) not null,
Cognome  varchar(6) not null,
...
unique (Cognome, Nome)
```

Si noti che è diverso dal caso:

```
Nome     varchar(6) not null unique,
Cognome  varchar(6) not null unique,
```

questo secondo è più restrittivo perché non solo la coppia `(Cognome,Nome)` ma anche i singoli `Cognome` e `Nome` non possono esser doppi.

Se il sistema rileva una violazione del vincolo, il comando di aggiornamento viene rifiutato, segnalando errore all'utente.

2.2.4.2 Column integrity constraint: `PRIMARY KEY`

SQL permette di specificare il vincolo `primary key` una sola volta per ogni tabella (a contrario di `unique` e `not null`). Anche questo può essere specificato su un singolo attributo o su più attributi, con sintassi speculare al caso `unique`. Se il sistema rileva una violazione del vincolo, il comando di aggiornamento viene rifiutato, segnalando errore all'utente.

2.2.4.3 Vincoli interrelazionali: `FOREIGN KEY`

Per la definizione di vincoli di integrità referenziale, in SQL si usa la keyword `foreign key`.
Il vincolo impone che per ogni riga della tabella interna il valore dell'attributo specificato, se diverso dal valore nullo, sia presente nelle righe della tabella esterna tra i valori del corrispondente attributo. L'unico requisito richiesto è che l'attributo cui si fa riferimento nella tavella esterna sia `unique`[1].
La sintassi si differenzia a seconda in una tabella vi sia uno o più attributi sotto vincolo di integrità referenziale:

- nel caso di un unico attributo, si può utilizzare `references`, come ad esempio in

  ```
  ...
  Dipart  varchar(15) references Dipartimento(NomeDip)
  ...
  ```

  facendo riferire l'attributo `Dipart` della tabella corrente a `NomeDip` della tabella `Dipartimento`;

- nel caso di due o più attributi si deve utilizzare `foreign key` come segue:

[1] Tipicamente l'attributo della tabella esterna cui si fa riferimento è la chiave primaria della stessa.

```
...
Nome    varchar(20),
Cognome varchar(20),
...
foreign key (Nome,Cognome) references Anagrafica(Nome,Cognome)
```

La corrispondenza tra gli attributi locali e quelli esterni avviene in base all'ordine: al primo attributo argomento di `foreign key` corrisponde il primo attributo argomento di `references`, e via via gli altri attributi.
Se dopo references la successione di colonne non è specificata si adotta la successione delle variabili adottate come chiave primaria (che deve esistere).

Il vincolo di integrità referenziale può esser violato:

- *operando* sulle righe della *tabella interna*: si inseriscono valori che non trovano riscontro nella tabella referenziata;
- operando su quelle della *tabella esterna*: si modifica o cancella un valore nella tabella referenziata cui erano legati record nella tabella interna, i quali non trovano più riscontro in un valore della referenziata.

Nel primo caso, inserimenti che comportano una violazione semplicemente non sono accettate dal sistema (lo stesso è avvenuto per gli altri vincoli visti sinora); nel secondo, invece, SQL permette di scegliere una eventuale reazione:

- `no action` (default): l'azione di modifica/cancellazione non viene consentita;
- `cascade`: nel caso di modifica il nuovo valore dell'attributo della tabella esterna viene riportato su tutte le corrispondenti righe della interna. Nel caso di cancellazione, tutte le righe della tabella interna corrispondenti alla riga cancellata vengono cancellate;
- `set null`: all'attributo che fa riferimento al valore modificato/cancellato viene assegnato il valore nullo;
- `set default`: all'attributo che fa riferimento al valore modificato/cancellato viene assegnato il valore di default.

La politica di reazione viene specificata immediatamente dopo il vincolo di integrità, secondo la seguente sintassi:

```
on < delete|update > < cascade|set null|set default|no action >
```

Ad esempio:

```
...
Dipart varchar(15) references Dipartimento(NomeDip)
                   on delete set null
                   on update cascade,
...
```

2.2.4.4 Vincoli generali con `CHECK`

`CHECK` è il modo più gnerale per creare controlli inter/intrarelazionali generali sui dati; inserimenti/modifiche che non rispettino i check impostati non sono rese possibili dal sistema.
La sintassi è:

```
CHECK (Condizione)
```

dove per `Condizione` può esser utilizzata una che possa apparire come argomento della clausola `where` di una `select`.
Può essere utilizzato:

- a livello di singola variabile (quindi formando column integrity constraint). Un esempio:

  ```
  CREATE TABLE PLAYERS_X
       (PLAYERNO INTEGER NOT NULL,
        SEX CHAR(1) NOT NULL CHECK(SEX IN ('M', 'F')))
  ```

  In questo caso l'espressione di `CHECK` può contenere solamente il nome della variabile considerata;

- a livello di tabella. Un esempio:

  ```
  CREATE TABLE PLAYERS_Z
    (PLAYERNO SMALLINT NOT NULL,
     BIRTH_DATE DATE,
     JOINED SMALLINT NOT NULL,
     CHECK(YEAR(BIRTH_DATE) < JOINED))
  ```

  Qui nella condizione di check si possono specificare più variabili della tabella; inoltre si noti che `NOT NULL` è un caso particolare (potrebbe esser sostituito da un `CHECK(COLUMN IS NOT NULL)`) ma che è meglio utilizzare direttamente poiché ottimizzato.

Roba forse non standard Mediante la clausola `check` è possibile specificare asserzioni[2], ovvero vincoli generici associati a politiche di controllo immediate o differite. Un'asserzione si crea mediante:

```
create assertion NomeAsserzione check (Condizione)
```

I vincoli:

- *immediati*, sono verificati immediatamente dopo ogni modifica della base di dati; se la condizione non è soddisfatta la base di dati torna alla propria istanza precedente la modifica (*rollback parziale*);
- *differiti*, impostati per le *transazioni*, sono verificati al termine dell'esecuzione della stesa. Il controllo differito viene tipicamente usato per gestire situazioni in cui non è possibile costruire una situazione consistente con una singola modifica alla base di dati. Se la condizione non è soddisfatta la base di dati torna alla propria istanza precedente l'intera transazione (*rollback*).

[2] Da verificare se presente questa cosa nel DBMS che stai usando?

La politica di controllo (immediata vs. differita) viene impostata mediante:

```
set constraints  [NomeVincoli | all]   < immediate | deferred >
```

2.2.4.5 Nominare i table constraint

Consigliato dare un nome ai constraint di tabella che si impone perché in sede di modifiche illegali verranno citati i nomi dei constraint che sono violati da un determinato inserimento/modifica.
Un esempio minimale:

```
CREATE TABLE PLAYERS
  (PLAYERNO INTEGER NOT NULL,
   SEX CHAR(1) NOT NULL,
   CONSTRAINT PRIMARY_KEY_PLAYERS  PRIMARY KEY(PLAYERNO),
   CONSTRAINT ALLOWED_VALUES_SEX CHECK(SEX IN ('M', 'F')))
```

Un esempio con sqlite3:

```
sqlite> insert into players values (1, 'A');
Error: CHECK constraint failed: ALLOWED_VALUES_SEX
```

2.3 Creazione di viste

2.3.1 Introduzione

In SQL è possibile creare tabelle definite come il risultato di una query; tali tabelle sono dette *viste* (mentre le tabelle ottenute per mezzo dello statement `CREATE TABLE` vengono dette *tabelle base*).
I dati in tutte le tabelle base sono fisicamente memorizzati; le viste, invece, possono essere considerate come tabelle "virtuali", cioè non vi è un dato memorizzato fisicamente e ad esse associato, bensì sono una rielaborazione di tabelle base.
Le applicazioni principali delle viste sono:

- semplificazione di operazioni frequenti;
- riorganizzazione di tabelle;
- gestione delle autorizzazioni all'accesso ai dati.

2.3.2 Definizione

La sintassi è:

```
<create view statement> ::=
  CREATE [ OR REPLACE ] VIEW <view name> [ <column list> ]
  AS <table expression>
  [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

O in una veste alternativa più compatta:

```
CREATE VIEW NomeVista [(column-name)] AS fullselect
```

- la `column-name` (`column list`) definisce i nomi colonna della vista; se specificata deve essere del numero di attributi risultato della fullselect; se non specificata la vista eredita i nomi delle colonne della tabella risultato della table expression
- la `column-name` deve essere specificata quando la tabella risultato della fullselect ha colonne non nominate (ad esempio definite da funzioni, costanti, espressioni e non rinominate con `AS`)

Un esempio di definizione:

```
CREATE VIEW CPLAYERS AS
SELECT PLAYERNO, LEAGUENO
FROM PLAYERS
WHERE LEAGUENO IS NOT NULL
```

Un esempio dove si sfrutta la column list

```
CREATE VIEW STRATFORDERS (PLAYERNO, NAME, INIT, BORN) AS
SELECT PLAYERNO, NAME, INITIALS, BIRTH_DATE
FROM   PLAYERS
WHERE  TOWN = 'Stratford'
```

2.3.3 Restrizioni sull'update di viste

E' possibile modificare le tabelle di base effettuando modifiche sulle viste: ad esempio

```
CREATE VIEW VETERANS AS
SELECT *
FROM   PLAYERS
WHERE  BIRTH_DATE < '1960-01-01'

UPDATE VETERANS
SET    BIRTH_DATE = '1970-09-01'
WHERE  PLAYERNO = 2
```

La modifica di sopra viene effettuata sulla tabella; incidentalmente la prossima volta che si accede alla vista il giocatore modificato non risulterà tra i suoi record.
Tuttavia vi sono restrizioni sulla possibilità di aggiornare i dati mediante una vista. Le più importanti sono:

- la `SELECT` che definisce la vista non deve contenere `DISTINCT`, funzioni di aggregazione, `GROUP BY` (ne `HAVING`), `ORDER BY`, operatori insiemistici
- la `FROM` clause non può contenere più di una tabella
- non si possono modificare colonne virtuali come ad esempio `BEGIN_AGE` nel caso seguente

  ```
  CREATE VIEW AGES (PLAYERNO, BEGIN_AGE) AS
  SELECT PLAYERNO, JOINED - YEAR(BIRTH_DATE)
  FROM   PLAYERS
  ```

2.3.4 Impostare check sugli aggiornamenti mediante vista

Si può voler impostare il vincolo che qualsiasi modifica a record non comporti la perdita/non visualizzazione degli stessi record dalla vista. Questo è fatto mediante `WITH CHECK OPTION`, che fa si che tutti i cambiamenti (`UPDATE`, `INSERT` e `DELETE`) siano controllati. Se impostato gli statement:

- `UPDATE`: la modifica è valida se il record modificato appartiene ancora alla vista in seguito;
- `INSERT`: l'inserimento è corretto se le nuove righe appartengono poi alla vista
- `DELTE`: la cancellazione è valida se le righe cancellate appartengono al contenuto della vista

Considerando poi che una vista può essere creata sulla base di un'altra vista abbiamo l'ulteriore possibilità di estendere il check:

- se qualifichiamo con `CASCADED` (comunque è il default) tutte le viste sono checkkate;
- se specofichiamo `LOCAL` il check è effettuato solamente sulla vista corrente.

2.4 Inserimento modifica e cancellazione dei dati

I comandi che permettono di modificare la base di dati sono `INSERT`, `UPDATE` e `DELETE`; sono caratterizzati da uno schema simile.

2.4.1 Inserire righe

La BNF di `INSERT` è:

```
<insert statement> ::=
  INSERT INTO <table specification> <insert specification>

<insert specification> ::=
  [ <column list> ] <values clause> |
  [ <column list> ] <table expression>

<column list> ::=
  ( <column name> [ , <column name> ]... )

<values clause> ::=
  VALUES <row expression> [ , <row expression> ]...

<row expression> ::=
 ( <scalar expression> [{, <scalar expression>}...]
```

Il comando di inserimento di righe nella base di dati presenta due sintassi alternative:

- la prima forma permette di inserire una o più righe all'interno delle tabelle; usata tipicamente all'interno dei programmi per riempire una tabella con i dati forniti direttamente dagli utenti (funge solitamente da *back-end* per una maschera di inserimento).
  Un esempio:

  ```
  insert into studente values
    (5, 'Gino', 'Neri', 'M', NULL, NULL),
    (6, 'Pina', 'Barocco', 'F', NULL, NULL)  ;
  ```

  Invece di specificare una `insert` per ogni record è meglio farne una unica: così si ha la garanzia che verranno inserite o tutte o nessuna;

- la seconda forma permette di aggiungere degli insiemi di righe estratti dal contenuto della base di dati (si veda la sezione sulle `select` per la sintassi di tale comando). Ad esempio

  ```
  insert into ProdottiMilanesi
      (select Codice, Descrizione
       from Prodotto
       where LuogoProd='Milano')
  ```

Si può specificare le colonne della dove andare a effettuare le modifiche (`column list`); quando non è specificata, viene implicitamente assunta la lista comprendente tutte le colonne della tabella, da sinistra a destra.
Se in un inserimento non vengono specificati i valori di tutti gli attributi, agli attributi mancanti viene assegnato il valore di default, o in assenza il valore nullo.
Si noti che l'ordine dei dati è importante: la corrispondenza tra gli attributi della tabella e i valori da inserire (o mediante inserimento manuale o mediante `select`) è data dall'ordine in cui compaiono i termini nella definizione della tabella.

2.4.2 Aggiornare righe

La BNF di `UPDATE` è:

```
<update statement> ::=
  UPDATE <table reference>
  SET <column assignment> [ , <column assignment> ]...
  [ <where clause> ]

<table reference> ::=
  <table specification> [ [ AS ] <pseudonym> ]

<pseudonym> ::= <name>

<column assignment> ::=
  <column name> = <scalar expression>
```

`UPDATE` permette di aggiornare uno o più attributi delle righe di una tabella che soddisfano l'eventuale condizione specificata con `where` (se non specificata, di default come soddisfatta per tutte le righe).
Il nuovo valore può esser:

- risultato di una *espressione*, la quale può far anche riferimento al valore corrente dell'attributo che verrà modificato;
- esito di una interrogazione SQL;
- `null`;
- valore di default.

Ad esempio per trasferire il dipendente 01 al dipartimento 3 e con salario incrementato del 10%:

```
UPDATE DIPENDENTI
SET NUMERO = 3, SALARIO = SALARIO*1.1
WHERE CODICE = '01'
```

Un esempio con l'uso di pseudonimo:

```
UPDATE PLAYERS AS P
SET    P.LEAGUENO = '2000'
WHERE  P.PLAYERNO = 95
```

Per aggiornare la quantità conservata dei prodotti ordinati sottraendo la quantità totale ordinata, ma solo per gli ordini posteriori al giugno 1999:

```
UPDATE PRODOTTI
SET QUANTITA = QUANTITA -
    ( SELECT SUM(QUANTITA)
      FROM DETTAGLI_ORDINE,ORDINI
      WHERE CODICE=PRODOTTI.CODICE
      AND DETTAGLI_ORDINE.NUMERO=ORDINI.NUMERO
      AND DATA > '1999-06-30' )
WHERE CODICE IN
    ( SELECT CODICE
      FROM DETTAGLI_ORDINE,ORDINI
      WHERE DETTAGLI_ORDINE.NUMERO=ORDINI.NUMERO
      AND DATA > '1999-06-30' )
```

2.4.3 Eliminare righe

La BNF di `DELETE` è:

```
<delete statement> ::=
  DELETE
  FROM <table reference>
  [ <where clause> ]

<table reference> ::=
  <table specification> [ [ AS ] <pseudonym> ]

<pseudonym> ::= <name>
```

`DELETE` elimina righe dalle tabelle della base di dati; quando la `where` non viene specificata, il comando cancella tutte le righe della tabella, altrimenti vengono rimosse solo le righe che soddisfano la condizione. Ad esempio per cancellare gli ordini dei prodotti con quantità ordinata maggiore della quantità conservata:

```
DELETE FROM DETTAGLI_ORDINE
WHERE QUANTITA >
   ( SELECT QUANTITA
     FROM PRODOTTI
     WHERE CODICE = DETTAGLI_ORDINE.CODICE )
```

2.5 Modifica ed eliminazione delle tabelle

2.5.1 Modifica

È possibile modificare la struttura di una tabella (anche qualora contenga già dei dati) mediante `alter table`:

- mediante `ADD` vengono aggiunte le colonne o i vincoli specificati;
- mediante `ALTER` si varia la dimensione di una colonna di tipo stringa a lunghezza variabile (può essere solo aumentata)
- mediante `DROP` le colonne e i vincoli specificati vengono distrutti. Nel caso di `PRIMARY KEY`, la cancellazione è estesa ai vincoli referenziali per i quali la corrente tabella è di riferimento.

Alcuni esempi di inserimento colonne:

```
ALTER TABLE studente ADD num_esami  INTEGER(2) ;
ALTER TABLE studente ADD iscrizione DATE AFTER nascita;
```

Esempio di eliminazione colonna:

```
ALTER TABLE studente DROP num_esami;
```

2.5.2 Eliminazione

Per eliminare una tabella indice, schema, trigger o vista si usa `drop`. Per una tabella:

```
drop table [if exists] <nome tabella>
```

L'effetto è di eliminare i dati (come in `delete`) ma anche lo schema della tabella e tutte le relazioni con gli altri elementi del db (link, privilegi ecc).
La specificazione di `if exists` fa si che nel caso in cui la tabella non esista non venga segnalato errore.
Per i rimanenti

```
drop index <nome>
drop schema <nome>
drop trigger <nome>
drop view <nome>
```

Se si distrugge una tabella o vista, viste e trigger che si riferiscono alla tabella sono marcati come non operativi.

2.6 Utenti e permessi

In SQL il controllo di accesso si fonda sui concetti di utente, password e privilegio; in particolare gli utenti possiedono dei privilegi di accesso alle risorse del sistema, che determinano cosa possono fare.
Quando una risorsa viene creata, il sistema concede automaticamente tutti i privilegi su tale risorsa al creatore; l'amministratore del db possiede tutti i privilegi su tutte le risorse.
L'utente proprietario di una risorsa è autorizzato a compiere su di essa qualsiasi operazione. Tuttavia un sistema in cui solo i proprietari fossero autorizzati a farne uso sarebbe di limitata utilità. Pertanto il proprietario di una risorsa può concedere uno o più privilegi di azione sulla stessa.

2.6.1 Gestione utenti

2.6.1.1 Creazione di utenti

La BNF per la creazione di utenti è:

```
<create user statement> ::=
  CREATE USER <user name> IDENTIFIED BY <password>

<user name> ;
<password>  ::= <alphanumeric literal>
```

Gli utenti, appena creati non hanno molti privilegi; possono loggarsi ed effettuare operazioni per le quali non sono richiesti privilegi (es listare i db, consultare l'help in linea ecc). Non possono creare risorse.

2.6.1.2 Modifica password

La modifica della password viene effettuata mediante `ALTER USER`

```
<alter user statement> ::= ALTER USER <user name> IDENTIFIED BY <password>
```

2.6.1.3 Rimozione di utenti

La rimozione di utenti viene effettuata mediante `DROP USER`:

```
<drop user statement> ::= DROP USER <user name>
```

2.6.2 Concedere privilegi

I privilegi concessi possono essere:

- di tabella:
- di database:
- di utente:

2.6.2.1 Privilegi di tabella

SQL supporta i seguenti privilegi di tabella:

- `SELECT`: da all'utente la possibilità di accedere alla tabella specificata con lo statemente omonimo; per le viste gli utenti debbono possedere i privilegi su tutte le tabelle di base
- `INSERT`: possibilità di inserire righe
- `DELETE`: possibilità di eliminare righe
- `UPDATE`: possibilità di modificare righe
- `REFERENCES`: possibilità di creare foreign keys che si riferiscano alla tabella considerata
- `ALTER`: possibilità cambiare la struttura della tabella con `ALTER TABLE`
- `INDEX`: possibilità di definire indici sulla tabella
- `ALL`: tutti i precedenti

La sintassi BNF è:

```
<grant statement> ::=
    <grant table privilege statement>    |
    <grant database privilege statement> |
    <grant user privilege statement>     |

<grant table privilege statement> ::=
  GRANT <table privileges>
  ON    <table specification>
  TO    <grantees>
  [ WITH GRANT OPTION ]

<table privileges> ::=
  ALL [ PRIVILEGES ] |
  <table privilege> [ , <table privileges> ]...

<table privileges> ::=
  SELECT                       |
  INSERT                       |
  DELETE                       |
  UPDATE     [ <column list> ] |
  REFERENCES [ <column list> ] |
  ALTER                        |
  INDEX

<column list> ::=
  ( <column name> [ , <column name> ]... )

<grantees> ::=
  PUBLIC |
  <user name> [ , <user name> ]...
```

2.6.2.2 Privilegi di database

Sono privilegi validi all'interno di un determinato database, le principali:

- `SELECT`, `INSERT`, `DELETE`, `UPDATE`, `REFERENCES`, `CREATE`, `ALTER`, `INDEX` funzionano analogamente a quanto vista nella sezione precedente, ma per tutte le tabelle del database di cui viene concesso permesso
- `DROP`: permesso di eliminare tabelle del database considerato
- `CREATE TEMPORARY TABLES`: come il nome suggerisce
- `CREATE VIEW`: come il nome suggerisce

La sintassi BNF:

```
<grant database privilege statement> ::=
  GRANT <database privileges>
  ON    [ <database name> . ] *
  TO    <grantees>
  [ WITH GRANT OPTION ]

<database privileges> ::=
  ALL [ PRIVILEGES ] |
  <database privilege> [ , <database privilege> ]...

<database privilege> ::=
  SELECT                    |
  INSERT                    |
  DELETE                    |
  UPDATE                    |
  REFERENCES                |
  CREATE                    |
  ALTER                     |
  DROP                      |
  INDEX                     |
  CREATE TEMPORARY TABLES   |
  CREATE VIEW               |
  SHOW VIEW                 |
  CREATE ROUTINE            |
  ALTER ROUTINE             |
  EXECUTE ROUTINE           |
  LOCK TABLES
```

2.6.2.3 Privilegi di utente

Di fatto sono gli stessi privilegi di database con l'aggiunta di `CREATE USER` e concedono ad un dato utente la possibilità di svolgere le azioni per/su qualsiasi database.
La sintassi BNF:

```
<grant user privilege statement> ::=
  GRANT <user privileges>
```

```
  ON    *.*
  TO    <grantees>
  [ WITH GRANT OPTION ]

<user privileges> ::=
  ALL [ PRIVILEGES ] |
  <user privilege> [ , <user privilege> ]...

<user privilege> ::=
  SELECT                    |
  INSERT                    |
  DELETE                    |
  UPDATE                    |
  REFERENCES                |
  CREATE                    |
  ALTER                     |
  DROP                      |
  INDEX                     |
  CREATE TEMPORARY TABLES   |
  CREATE VIEW               |
  SHOW VIEW                 |
  CREATE ROUTINE            |
  ALTER ROUTINE             |
  EXECUTE ROUTINE           |
  LOCK TABLES               |
  CREATE USER
```

2.6.2.4 Estendere privilegi (`WITH GRANT OPTION`)

Se ai `GRANT` di cui prima viene specificata `WITH GRANT OPTION` si permette all'utente che riceve i privilegi concessi di estenderli/darli a sua volta.

2.6.2.5 Ruoli

SQL-3 (non supportato ovunque) ha introdotto un modello di controllo dell'accesso basato sui ruoli. Il *ruolo* è una sorta di contenitore di privilegi; in ogni istante un utente dispone dei privilegi che gli sono stati attribuiti direttamente e dei privilegi associati al/ai ruoli ricoperti.

Per la creazione e utilizzo di un ruolo:

```
<create role statement> ::=
  CREATE ROLE <role name>

<grant role statement> ::=
  GRANT <role name [ {,<role name>}... ]
  TO <grantees>
```

la definizione dei contenuti del ruolo avviene normalmente mediante `GRANT`; un esempio completo:

```
CREATE ROLE SALES;
```

```
GRANT  SELECT, INSERT
ON     PENALTIES
TO     SALES;

GRANT SALES TO ILENE, KELLY, JIM, MARC;
```

2.6.3 Revocare privilegi

Lo statement `REVOKE` può essere usato dall'utente che aveva concesso in precedenza i privilegi per toglierli

```
<revoke statement> ::=
  <revoke table privilege statement>   |
  <revoke database privilege statement |
  <revoke user privilege statement>    |
  <revoke role statement>

<revoke table privilege statement> ::=
  REVOKE <table privileges>
  ON     <table specification>
  FROM   <grantees>

<table privileges> ::=
  ALL [ PRIVILEGES ] |
  <table privilege> [ , <table privilege> ]...

<table privilege> ::=
  SELECT                       |
  INSERT                       |
  DELETE                       |
  UPDATE     [ <column list> ] |
  REFERENCES [ <column list> ] |
  CREATE                       |
  ALTER                        |
  INDEX                        |
  DROP

<revoke database privilege statement> ::=
  REVOKE  <database privileges>
  ON      [ <database name> . ] *
  FROM    <user name> [ , <user name> ]...

<database privileges> ::=
  ALL [ PRIVILEGES ] |
  <database privilege> [ , <database privilege> ]...

<database privilege> ::=
  SELECT                  |
  INSERT                  |
```

```
  DELETE                  |
  UPDATE                  |
  REFERENCES              |
  CREATE                  |
  ALTER                   |
  DROP                    |
  INDEX                   |
  CREATE TEMPORARY TABLES |
  CREATE VIEW             |
  SHOW VIEW               |
  CREATE ROUTINE          |
  ALTER ROUTINE           |
  EXECUTE ROUTINE         |
  LOCK TABLES

<revoke user privilege statement> ::=
  REVOKE <user privileges>
  ON     *.*
  FROM   <user name> [ , <user name> ]...

<user privileges> ::=
  ALL [ PRIVILEGES ] |
  <user privilege> [ , <user privilege> ]...

<user privilege> ::=
  SELECT                    |
  INSERT                    |
  DELETE                    |
  UPDATE                    |
  REFERENCES                |
  CREATE                    |
  ALTER                     |
  DROP                      |
  INDEX                     |
  CREATE TEMPORARY TABLES   |
  CREATE VIEW               |
  SHOW VIEW                 |
  CREATE ROUTINE            |
  ALTER ROUTINE             |
  EXECUTE ROUTINE           |
  LOCK TABLES               |
  CREATE USER


<column list> ::=
 (<column name> [ {, <column name>}...])

<revoke role statement> ::=
  REVOKE <role name> [ {, <role name>}...]
  FROM   <grantees>
```

```
<grantees> ::=
  PUBLIC |
  <user name> [ {, <user name>}...]
```

2.7 Schema

SQL consente la definizione di uno schema di dati ovvero una collezione di oggetti (tabelle, viste) cui si da un nome comune per motivi di organizzazione logica.
Una precisazione sui nomi/identificatori usati sin'ora:

- i nomi degli oggetti (tabelle, viste) si esprimono in due forme: con un unico identificatore (come fatto sin'ora), oppure costituiti da due identificatori separati da un punto. In quest'ultimo caso il primo identificatore è il nome dello schema di appartenenza.
- la omissione del nome di schema equivale alla assunzione implicita del nome dello *schema corrente*;
- lo schema corrente può essere cambiato con `SET SCHEMA`.
  Inizialmente, il nome dello schema corrente coincide col nome dell'utente.

Ogni utente ha quindi un proprio schema ad esso associato che negli statement di creazione è solitamente sottointeso, ma che potrebbe esser esplicitato.

2.7.1 Creazione di schema

La definizione di un nuovo schema avviene mediante la sintassi

```
<create schema statement> ::=
  CREATE SCHEMA <schema name> <schema statement>...

<schema statement> ::=
  <create table statement> |
  <create view statement>  |
  <create index statement> |
  <grant statement>
```

Un esempio:

```
CREATE SCHEMA TWO_TABLES
  CREATE TABLE TABLE1 (COLUMN1 INTEGER)
  CREATE TABLE TABLE2 (COLUMN1 INTEGER)
  CREATE VIEW ... (ecc)
```

2.7.2 Rimozione di uno schema

Per la rimozione:

```
<drop schema statement> ::=
  DROP SCHEMA <schema name> [RESTRICT]
```

Se lo schema contiene elementi anche essi vengono eliminati, a meno che non si specifichi `RESTRICT`; nel qual caso l'eliminazione è limitata allo schema.

2.7.3 Differenze di schema vs utente

Il concetto di schema è simile a quello di owner e se ogni owner di fatto è un SQL user ci si chiederò l'utilità di introdurre il concetto di schema.
Uno schema ed un utente sql hanno funzionalità overlappanti, ma anche feature distintive; le peculiarità di uno schema rispetto ad un user:

- uno schema non ha password (a differenza dell'utente)
- non è possibile loggarsi con uno schema name
- non è possibile assegnare privilegi ad uno schema con `GRANT`

2.8 Trigger

I trigger servono per introdurre automazione nei dbms, che sono passivi per natura, ovvero effettuano azioni solo se viene chiesto loro.
Il paradigma è *Evento-Condizione-Azione*; quando si verifica l'evento, se la condizione è soddisfatta, allora viene svolta l'azione:

- eseguite quando occorre una specifico *evento* di inserimento (`INSERT`), cancellazione (`DELETE`) o aggiornamento (`UPDATE`) su una tabella;
- la *condizione* è un predicato booleano espresso in SQL
- l'*azione* è una sequenza di primitive SQL (talvolta arricchite da un linguaggio di programmazione, a seconda dell'ambiente)

I trigger hanno due **livelli di granularità**:

- *row level*: l'attivazione avviene per ogni tupla coinvolta nell'operazione (comportamento orientato alle singole istanze);
- *statement level*: l'attivazione avviene una sola volta per ogni primitiva SQL facendo riferimento a tutte le tuple coinvolte dalla primitiva (comportamento orientato agli insiemi).

I trigger possono avere **modalità** *immediata* o *differita*:

- *immediata*: la loro valutazione in genere avviene immedia tamente dopo l'evento che li ha attivati (opzione `after`), o più raramente prima dell'evento cui si riferisce (`before`)
- *differita*: la valutazione avviene alla fine della transazione, a seguito di un comando `commit work`

2.8.1 Definizione

La sintassi

```
<create trigger statement> ::=
  CREATE TRIGGER <trigger name>
   <trigger moment>
   <trigger event>
```

```
   [<trigger condition>]
   <trigger action>

<trigger moment> ::= BEFORE | AFTER | INSTEAD OF

<trigger event> ::=
  { INSERT | DELETE | UPDATE [OF <column list>] }
  { ON | OF | FROM | INTO }  <table specification>
  [ REFERENCING {OLD | NEW | OLD_TABLE | NEW_TABLE} AS <variable>]
  FOR EACH { ROW | STATEMENT }

<trigger condition> ::= (WHEN <condition> )

<trigger action> ::= <begin-end block>
```

- `trigger moment` specifica quando il trigger viene eseguito in relazione alle azioni che l'hanno attivato. Basicamente è un `BEFORE` o `AFTER`. In alcuni prodotti si può specificare `INSTEAD OF` (l'azione scatenante non viene di fatto eseguita, solamente il trigger lo è) oppure `NO CASCADE BEFORE` (il no cascade serve per evitare che le azioni del trigger non attivino altri trigger in cascata);
- `trigger event` specifica quali azioni fanno partire il trigger: si tratta basicamente di `INSERT`, `DELETE` o `UPDATE`.
  `ON`, `OF`, `FROM`, `INTO` possono essere utilizzate intercambiabilmente.
  Dopo `FOR EACH` si esprime la granularità e può essere `row` (tupla) oppure `statement` (primitiva).
  La clausola opzionale `REFERENCES` consente di introdurre nomi per designare le singole righe interessate prima (`OLD AS`) o dopo l'operazione (`NEW AS`) oppure l'intera collezione di righe interessate, prima o dopo (`OLD_TABLE AS`, `NEW_TABLE AS`);
- `trigger condition`: `WHEN` introduce una condizione che deve essere verificata per la effettiva esecuzione delle azioni.

Alcuni esempi Per segnalare una errore quando l'aumento di salario di un dipendente supererebbe il 10%:

```
CREATE TRIGGER LIMITE_AUMENTO
  AFTER UPDATE OF SALARIO ON DIPENDENTI
  REFERENCING NEW AS N OLD AS O
  FOR EACH ROW MODE DB2SQL
  WHEN (N.SALARIO > 1.1 * O.SALARIO)
    SIGNAL SQLSTATE '75000' ('Incremento salario > 10%')
```

Segnalare un errore quando, inserendo un nuovo ordine per un prodotto, la quantità totale ordinata per quel prodotto sarebbe superiore alla quantità disponibile:

```
CREATE TRIGGER CONTR_DISP
  NO CASCADE BEFORE INSERT ON DETTAGLI_ORDINE
  REFERENCING NEW AS ORD_PROD
```

```
  FOR EACH ROW MODE DB2SQL
  WHEN ( (SELECT SUM(QUANTITA)+ORD_PROD.QUANTITA
          FROM DETTAGLI_ORDINE
          WHERE CODICE=ORD_PROD.CODICE) >
         (SELECT QUANTITA
          FROM PRODOTTI
          WHERE ORD_PROD.CODICE=CODICE) )
SIGNAL SQLSTATE '75001'
  ('Disponibilita‘ insufficiente per l''ordine');
```

2.8.2 Rimozione

Per la rimozione di trigger

```
<drop trigger statement> ::=
  DROP TRIGGER [ <table name> . ] <trigger name>
```

Parte II

DBMS

Capitolo 3

Sqlite

3.1 Introduzione

Sqlite è una libreria C che fornisce un motore sql senza necessità di server o configurazioni; i database sono elaborati in memoria o possono esser salvati su disco come semplici file standalone (binari), aventi estensione `.db`.
È disponibile un client che fa uso della libreria e permette di simulare le funzionalità di un server SQL con requisiti minimi.

Installazione Per installare il client sqlite in Debian:

```
apt-get install sqlite3 sqlite3-doc
```

Per un tool grafico si utilizzi

```
apt-get install sqlitebrowser
```

Primo avvio `sqlite3` può essere invocato

- da solo mediante

  ```
  sqlite3
  ```

- opzionalmente indicando il nome di un database da aprire:

  ```
  sqlite3 mydata.db
  ```

  In questo caso, se il database non esiste verrà creato.

In seguito è già possibile impartire dei comandi sql.

Meta-comandi utili Il client `sqlite3` mette a disposizione una serie di meta-comandi, prefissati da un punto. Per listarli si comandi `.help`. I comandi più utili sono:

- `.open filename`: chiude il database corrente e apre il database del file specificato

- `.read filename`: esegui il file `.sql` specificato
- `.save filename`: salva il database in memoria nel file specificato
- `.tables`: lista le tabelle presenti nel database attuale;
- `.schema`: dumpa lo schema delle tabelle in sql;
- `.mode` specifica come deve essere visualizzato l'output; di base si utilizzi `column`, ma può essere utile anche `insert`, che produce un dump sql dell'oggetto richiesto, ad esempio

```
sqlite> .mode insert new_table
sqlite> select * from tbl1;
INSERT INTO "new_table" VALUES('hello',10);
INSERT INTO "new_table" VALUES('goodbye',20);
sqlite>
```

- `.output filepath` per redirezionare l'output permanentementente su un file, mentre `.once filepath` redireziona solo l'output del prossimo comando sul file, tornando su standard output in seguito
- `.dump file` fa il dump dell'intero database in sql; per utilizzarlo in uno script si potrebbe fare

```
echo '.dump' | sqlite3 miodb.db > miodb.sql
```

Configurazione Il file `/.sqliterc` è il file di configurazione letto allo startup dal client; i meta comandi dati al suo interno verranno utilizzati per preparare l'ambiente interattivo del client.
Alcune opzioni che molto probabilmente si desiderano sono:

```
.header on
.mode column
```

Importazione di `csv` L'importazione di un csv tipo

```
1,2
5,6
2,7
```

avviene mediante, tipicamente:

```
sqlite> create table foo(a, b);
sqlite> .mode csv
sqlite> .import test.csv foo
```

Capitolo 4

PostgreSQL

4.1 Primo setup

Avviato il server, occorre connettersi come amministratore; il login da amministratore in postgres è rappresentato dall'user `postgres`

```
su                  # switch user a root
su postgres         # switch user a postgres
psql
```

Per creare un utente con password

```
postgres=# CREATE USER l ENCRYPTED PASSWORD 'password';
```

Se si vuole concedere la possibilità di creare database
Creiamo poi un database per un utente

```
postgres=# CREATE DATABASE test OWNER l;
```

4.2 Il client `psql`

Il client di default disponibile con Postgresql è psql; la configurazione dell'ambiente psql può esser fatta nel file `.psqlrc`, in cui si impartiscono i meta-comandi da dargli in inizializzazione.

4.2.1 Utilizzo interattivo e batch

Utilizzo interattivo Per listare i database disponibili

```
psql -l
```

Per loggarsi ad un database si specifica prima il nome poi l'utente con il quale ci si logga:

```
psql test l
```

in assenza di utente verrà utilizzato l'utente di sistema; dopo la richiesta della password ci si loggherà al server e si potrà immetter comandi.

Esecuzione batch di script `sql` Per l'esecuzione del file `batch.sql` come `l` sul db `test` abbiamo due possibilità :

- dalla shell

```
psql test l < batch-file.sql > output.txt
```

- dall'interno di `psql` mediante

```
\i batch-file.sql
```

4.2.2 Ottenere aiuto

Avviato il client in maniera interattiva, si può ottenere aiuto per:

- le sintassi SQL: mediante `\h` (per ottenere una lista di man) o `\h COMANDO` (per il manuale di un comando)
- i comandi di `psql`: si guardi `man psql` o `\?`.

4.2.3 Listare tabelle e strutture

Dopo aver creato differenti tabelle all'interno di una base di dati, due comandi risultano utili:

- `\d` per listare tutte le tabelle della base di dati

```
test=> \d ;
          Lista delle relazioni
 Schema | Nome  |  Tipo   | Proprietario
--------+-------+---------+--------------
 public | prova | tabella | l
```

  Nell'esempio la base contiene una tabella di nome `prova`

- `\d oggetto` per avere una descrizione dell'oggetto (tabella o altro)

```
test=> \d prova
      Tabella "public.prova"
 Colonna |  Tipo   | Modificatori
---------+---------+--------------
 x       | integer |
 y       | integer |
```

4.2.4 Input e output dati

Per immettere/salvare dati da/in un file testuale si utilizza il comando `COPY`. Detto comando SQL è però concesso solo al root del server; gli utenti normali possono utilizzare il comando `\copy` di `psql`, che presenta sintassi analoga (si differenzia nel fatto che la prima keyword è `\copy` invece che `COPY`).
Consideriamo le seguenti tre situazioni:

- input i record di un csv in una tabella già creata;

- output di una intera tabella su csv;
- output di una select.

Consideriamo il csv `dati.csv` così strutturato da:

```
"x";"y"
1;2
3;4
5;6
9;10
11;12
13;14
15;16
```

Per salvare detti record nella tabella `prova`:

```
\copy prova FROM 'dati.csv' DELIMITER ';' CSV HEADER QUOTE '"'
```

Per salvare una *tabella* su csv separato da virgole

```
\copy prova TO 'backup.csv' DELIMITER ',' CSV HEADER QUOTE '"'
```

Se invece si desidera salvare gli esiti di una *query*:

```
\copy (select * from prova where y>5) TO 'backup.csv' DELIMITER ',' CSV HEADER QUOTE '"'
```

4.2.5 Dump/restore

Per effettuare il *dump*, nella shell del sistema operativo comandare

```
pg_dump -U l test > test.sql
```

permette di effettuare il dump del db `test`, connettendosi come l'utente `l` nel file `test.sql`.
Per effettuare il *restore* della suddetta (es in un altro pc) comandare:

```
psql test l < test.sql
```