Cheat sheet SQL

Interrogazione

```
query ::=
  | SELECT [DISTINCT] attr-expr [[AS] id], ...
      from [WHERE cond] [order-by]
  | SELECT [attr | agfun([DISTINCT] attr)], ...
      from [WHERE cond]
      [ GROUP BY attr, ... [HAVING cond] ]
      [order-by]
  | query setop [ALL] query

attr-expr  ::=
  | attr-expr [AS] id
  | [ table "." ] "*"
  | attr | const
  | [uop] attr-expr [ bop attr-expr ]

table-as   ::= table [[AS] id]
theta-join ::= [ LEFT | RIGHT | FULL ] JOIN
from ::=
  | FROM table-as, ...
  | FROM table-as theta-join table-as
      ( ON cond | USING attr, ... )
  | FROM table-as NATURAL JOIN table-as

agfun ::= SUM | COUNT | AVG | MAX | MIN

order-by ::= ORDER BY attr [ ASC | DESC ], ...

setop ::= UNION | INTERSECT | EXCEPT
```

Se non è specificato `DISTINCT`, il risultato è una tabella che ammette duplicati — operazioni su multiinsiemi.

Join

```
SELECT * FROM A JOIN B ON A.x > B.y
SELECT * FROM A JOIN B USING x
```

Join su più tabelle si possono implementare manualmente:

```
SELECT * FROM A, B, C
WHERE A.x = B.y AND B.y = C.z
```

Aggregazione

```
SELECT x, MAX(x)
FROM A
GROUP BY x, y HAVING MIN(y) > 10
```

SELECT solo attributi anche nel `GROUP BY`

COUNT attributi distinti, o # righe con `COUNT(*)`

HAVING condizione con funzioni di aggregazione, alcuni DBMS non consentono subquery

Ordinamento

Se non specificato altrimenti, l'ordine è crescente.

Operazioni insiemistiche

```
SELECT x from A
UNION
SELECT y AS x from B
```

- `UNION`, `INTERSECT` ed `EXCEPT` (differenza)
- le due tabelle devono essere omogenee
- se non è specificato `ALL`, i duplicati vengono eliminati dal risultato

Condizioni

```
subquery ::= "(" query ")"
subquery-or-list ::= subquery | "(" val, ... ")"
atom ::= attr | val
cmp  ::= < | = | > | <> | <= | >=
cond ::=
  | atom cmp atom
  | cond ( AND | OR | NOT ) cond
  | atom IS [NOT] NULL
  | atom BETWEEN atom AND atom
  | [NOT] EXISTS subquery
  | attr cmp ( ANY | ALL ) subquery-or-list
  | atom [NOT] IN subquery-or-list
  | atom LIKE pat [ ESCAPE char ]
```

EXISTS vero se la subquery restituisce almeno una riga

ANY / **ALL** confronto vero per uno/tutti i valori restituiti dalla subquery di colonna

IN `ANY` dove *cmp* è l'uguaglianza

LIKE pattern matching, *pat* è una stringa in cui `%` corrisponde alla regex `.*`, `_` a `.` e il carattere indicato con `ESCAPE` a `\`

Subquery

Si distinguono subquery scalari (restituiscono un solo valore), di colonna e di tabella.

Possono utilizzare variabili legate dalla query principale:

```
SELECT * FROM A
WHERE EXISTS (SELECT * FROM B WHERE A.x = B.y)
```

Utile in particolare per `EXISTS`, che altrimenti è sempre vero o falso. Tuttavia, in questo modo la subquery viene eseguita per ogni ennupla della query principale; a volte si può migliorare l'efficienza passando da `EXISTS` a `IN` con subquery eseguita una sola volta.

Manipolazione dei dati

```
manipulation ::=
  | INSERT INTO table [ "(" attr, ... ")" ]
    ( VALUES "(" val, ... ")" | query )
```

```
  | DELETE FROM table
    [WHERE  cond]

  | UPDATE table
    SET attr = expr
    [ WHERE cond ]
```

Inserimento

```
INSERT INTO A (x, z)      INSERT INTO A
VALUES ('foo', 42)        SELECT x, y, z
                          FROM B
```

- i campi non specificati prendono `NULL` o un valore di default
- errore se non vengono specificati valori che non possono essere nulli (in particolare la chiave primaria)

Cancellazione

- se non è specificata la condizione cancella tutte le righe della tabella (lo schema rimane)
- spesso la condizione è sulla chiave primaria

Aggiornamento

```
UPDATE A
SET x = x * 2
WHERE y < 4
```

Definizione

```
definition ::=
  CREATE TABLE table "("
    attr-spec, ...
    [constr-name] table-constr, ...
  ")"

attr-spec ::=
  attr type [ DEFAULT const ]
  [ [constr-name] attr-constr ]

constr-name ::= CONSTRAINT constr-name

attr-constr ::=
  | NOT NULL | UNIQUE | PRIMARY KEY
  | CHECK "(" expr ")"
  | REFERENCES table "(" attr ")"

table-constr ::=
  | ( NOT NULL | UNIQUE | PRIMARY KEY ) "(" attr ")"
  | CHECK "(" expr ")"
  | FOREIGN KEY "(" attr, ... ")"
    REFERENCES table "(" attr, ... ")"
    [ ON ( DELETE | UPDATE )
      ( NO ACTION | CASCADE
      | SET NULL | SET DEFAULT ) ]

type ::=
  | INTEGER | CHAR(len) | VARCHAR(maxlen) | DATE
  | REAL | NUMBER(digits, decimals) | FLOAT(mantissa)
```

Vincoli

```
CREATE TABLE T (
  pk CHAR(10) PRIMARY KEY
  fk1 INTEGER
  fk2 INTEGER
  n INTEGER DEFAULT 0 CHECK (n >= 0)

  FOREIGN KEY (fk1, fk2)
  REFERERNCES S(a, b)
)
```

UNIQUE definisce una chiave, su più attributi la combinazione deve essere unica (non i singoli)

PRIMARY KEY una sola, implica `UNIQUE` e `NOT NULL`

FOREIGN KEY `ON DELETE` / `ON UPDATE` specificano cosa fare se l'ennupla riferita viene cancellata/modificata, `NO ACTION` (default) impedisce l'operazione

CHECK espressione booleana, se indicato come vincolo di attributo può riguardare solo quello

Se il vincolo riguarda solo una colonna viene specificato insieme alla definizione dell'attributo corrispondente, altrimenti è inserito come vincolo di tabella.

Viste

Tabelle virtuali che riorganizzano i dati di quelle fisiche. Scopi: presentazione semplificata, gestione dei permessi di visualizzazione/modifica, conversione unità/formati, indipendenza logica.

```
view-def ::=
  CREATE VIEW view [ "(" attr, ... ")" ] AS query
  [ WITH [ LOCAL | CASCADED ] CHECK OPTION ]
```

- la lista di attributi permette di rinominare le colonne risultanti dalla query
- la query non può contenere `INTERSECT`, `EXCEPT` (implementabili con `SELECT`), `UNION`, `GROUP BY` (che però si può usare nelle interrogazioni sulla view)
- le viste sono dinamiche sui risultati della subquery, ma non sulla struttura delle tabelle: data la vista

  ```
  CREATE VIEW V AS SELECT * FROM T
  ```

  se a `T` viene aggiunta una colonna, questa non sarà visibile in `V`

Viste di gruppo

Sono quelle che contengono `GROUP BY` o sono fanno riferimento ad una vista di gruppo.

- richiedono nomi di colonna espliciti
- non consentono la modifica di attributi corrispondenti a funzioni di aggregazione

```
CREATE VIEW V (a, s) AS
  SELECT a, SUM(b)
  FROM T
  GROUP BY a
```

Modificabilità

Con alcune eccezioni (e.g. subquery su più tabelle, funzioni di aggregazione), le viste sono modificabili. Si possono specificare i seguenti vincoli:

WITH [CASCADED] CHECK OPTION l'inserimento fallisce se la nuova ennupla non fa parte della vista

WITH LOCAL CHECK OPTION se la vista è definita in termini di altre viste, verifica solo le condizioni di quelle che specificano `WITH CHECK OPTION`, aziché di tutte come con `CASCADE`

Manipolazione dello schema

```
drop  ::= DROP TABLE table ( RESTRICT | CASCADE )
alter ::= ALTER TABLE table alter-cmd

alter-cmd ::=
  | ADD [COLUMN] attr-spec
      [ FIRST | AFTER attr ]
  | DROP [COLUMN] attr ( RESTRICT | CASCADE )

  | MODIFY [COLUMN] attr-spec
  | ALTER [COLUMN] attr
      ( SET DEFAULT const | DROP DEFAULT )

  | ADD CONSTRAINT constr-name table-constr
  | DROP CONSTRAINT constr-name
      ( RESTRICT | CASCADE )

  | DROP VIEW view ( RESTRICT | CASCADE )
```

ADD come ultima colonna se non specificato altrimenti

DROP `RESTRICT` / `CASCADE` specificano la gestione di vincoli di integrità referenziale invalidati

ADD CONSTRAINT nome obbligatorio

DROP CONSTRAINT `RESTRICT` / `CASCADE` usati in caso di rimozione di un vincolo di unicità o chiave primaria se ci sono chiavi esterne riferite alla colonna

DROP VIEW `RESTRICT` / `CASCADE` cosa fare se una vista dipende da quella da cancellare

L'aggiuna di una colonna `NOT NULL` si fa in 3 passaggi creandola senza vincolo, inserendo i dati e aggiungendo il vincolo.

Procedure e trigger

Procedure

```
proc-def   ::= CREATE FUNCTION proc IS
                 decls body
proc-decls ::= DECLARE ( var type ) ...
proc-body  ::= BEGIN proc-op ... END

proc-op ::=
  | SELECT query-attrs INTO var query-rest
  | RETURN "(" var ")"
  | manipulation
```

Trigger

Azione eseguita automaticamente al verificarsi di un determinato evento. Consentono di esprimere vincoli di integrità complessi, gestire copie ridondanti o ricavabili dei dati, e limitare le operazioni ammissibili.

```
trigger-def ::=
  CREATE TRIGGER trigger
  ( BEFORE | AFTER )
    ( INSERT | DELETE | UPDATE ), ...
  ON table
  [ FOR EACH ROW ]
  [ WHEN cond ]
  proc-decls proc-body
```

`FOR EACH ROW` determina se il trigger viene eseguito una volta per operazione o per record modificato.
Categorie di trigger:

attivi modificano lo stato della base di dati

passivi fanno fallire l'operazione

INSTEAD OF sostituiscono l'operazione

Controllo degli accessi

Ciascuna risorsa (tabella, attributo, vista...) ha un proprietario, che può assegnare privilegi su quella risorsa ad altri utenti.

```
privilege ::=
  | SELECT | DELETE
  | ( INSERT | UPDATE | REFERENCES )
      [ "(" attr, ... ")" ]

privileges ::= ( privilege, ... ) | ALL PRIVILEGES

grant ::=
  GRANT privileges ON resource
  TO user, ... [ WITH GRANT OPTION ]

revoke ::=
  REVOKE privileges ON resource
  FROM user, ... [ RESTRICT | CASCADE ]
```

INSERT gli attributi non specificati prendono `NULL`

REFERENCES creare una chiave esterna riferita agli attributi

WITH GRANT OPTION l'utente può propagare il privilegio ad altri

RESTRICT/CASCADE significativi se l'utente ha propagato il privilegio

Indici

```
index-def ::=
  CREATE INDEX index
  ON table "(" attr, ... ")"
```

Transazioni

`BEGIN TRANSACTION`, `COMMIT`, `ROLLBACK`