Foundations of Computer Science - SQL

1 Relational Databases

1.1 Introduzione

I database relazionali sono organizzati in tabelle che possono essere collegate tra di loro. Una tabella in un modello relazionale rappresenta, di per sé, una relazione. Per esempio, nella seguente tabella, ogni riga descrive un determinato libro:

ISBN	Title
978-1-449-30321-1	*Scaling MongoDB*
978-1-491-93200-1	*Graph Databases*
978-1-449-39041-9	*Cassandra: The Definitive Guide*
007-709500-6	*Database Systems*

Table 1: Some sample books

I dati in un modello relazionale sono organizzati in colonne e ogni entry (*cella*) contiene un **SINGOLO** dato; quindi il principale problema del modello relazionale sussiste nel maneggiare i dati con una molteplicità di elementi in una singola colonna, per esempio un libro con due autori.

La soluzione proposta dal modello relazionale consiste nel collegare due tabelle, ma nel fare ciò abbiamo bisogno di un ID di identificazione per ogni riga. Qui di seguito vediamo un esempio di due tabelle collegate.

book_id	ISBN	Title
1	978-1-449-30321-1	*Scaling MongoDB*
2	978-1-491-93200-1	*Graph Databases*
3	978-1-449-39041-9	*Cassandra: The Definitive Guide*
4	007-709500-6	*Database Systems*

Tabella 2: Books Table with ID

author_id	Name	Surname
1	Ian	Robinson
2	Kristina	Chodorow
3	Riccardo	Torlone
4	Paolo	Atzeni
5	Stefano	Ceri
6	Stefano	Paraboschi
7	Eben	Hewitt

Tabella 3: Authors Table with ID

Per connettere queste due tabelle usiamo una tabella di “relazione” o di “join”. Una tabella di “join” appare come mostrato di seguito:

book_id	author_id
2	1
3	7
1	2
4	4
4	5
4	6
4	3

Tabella 4: BookAuthors Table

Nella Tabella 4 si può notare la necessità dell’utilizzo di un ID (chiamato *Foreign Key*) per identificare ogni riga, così che ci si possa riferire alla stessa in maniera univoca. In alcuni casi, potremmo non avere bisogno di una terza tabella (che poi è la “join table”): uno di questi casi è dato dalla relazione “One-to-Many”. Anche la relazione One-to-Many richiede che le righe in una tabella abbiano ID unici, ma a differenza della tabella join usata nella relazione Many-to-Many, la tabella con la parte di dati relativi al lato “Many” (the many side) ha una colonna riservata agli ID della parte di dati relativi al lato “One”.

Gli ID usati per identificare univocamente le righe descritte nelle tabelle sono chiamate “*Primary Keys*” (**PK**). Se questa chiave primaria è usata in un’altra tabella è chiamata “*Foreign Key*” (**FK**) e di solito non risulta essere unica nella nuova tabella. Lo scopo della foreign key è quello di collegare le due tabelle in una relazione one-to-many.

Di solito il processo generativo di una chiave primaria è lasciato al Relational Database Management System (RDBMS) (è la strada più sicura da percorrere), che automaticamente genera la chiave e di solito risulta essere un ordinario numero intero. Per le tabelle di tipo join, la chiave primaria è una combinazione di foreign keys. Una chiave primaria composta da più di un attributo è chiamata “Composite Primary key”, o “Chiave Primaria composta” (CPK).

1.2 Forme Normali (Normal Forms)

La Prima Forma Normale (1NF) dice che i tipi devono essere *atomici* (atomic) (e cioè che la tabella deve essere piatta) così che non è possibile avere una cella con dati multipli (multi-list) al suo interno. Questo causa alcune anomalie, come ad esempio:

i) Ridondanza (Redundancy): Gli stessi dati sono ripetuti più di una volta. Per esempio, in un orario universitario (useremo questo esempio per l’intera lista di anomalie), se tutti i corsi sono tenuti esclusivamente in un’aula, avremo due attributi: uno con il Corso e l’altro con l’Aula, che è ridondante.
ii) Anomalia nell’aggiornamento (Update anomaly): quando si aggiorna un valore di una cella. Per esempio, se aggiorniamo il nome di un’aula in una cella non abbiamo un aggiornamento automatico su tutte le celle contenenti quell’aula o il corso relativo alla stessa;
iii) Anomalia nell’eliminazione (Delete anomaly): se tutti i dati che utilizzano una specifica informazione vengono eliminati (dropped), allora perderemo anche quella specifica informazione. Per esempio, se viene eliminata da tutti una lezione perdiamo l’informazione in merito all’aula in cui si tiene la lezione;
iv) Anomalia nell’inserimento (Insert anomaly): non possiamo utilizzare una specifica informazione senza accertarci che altri dati la utilizzino. L’esempio è simile a quello del punto iii): non possiamo prenotare una stanza per un corso senza sapere quanti studenti parteciperanno.

Per superare queste anomalie sono state create altre Forme Normali. Ma prima di introdurle, dobbiamo introdurre una definizione.

Def. Dipendenza funzionale (*functional dependency*):
Siano A, B due set di attributi, diciamo che A → B o che A determina funzionalmente B se, per ogni tupla t_1 e t_2:

$$t_1[A] = t_2[A] \Rightarrow t_1[B] = t_2[B]$$

e chiameremo proprio A → B una **dipendenza funzionale**.

In altre parole, ogni qual volta due tuple sono d'accordo su A, saranno d'accordo anche su B. Una dipendenza funzionale è quindi una forma di costrizione.

Nota bene. È molto facile confutare una dipendenza funzionale, ti basta una istanza che la violi, ma per dimostrarla è necessario verificare ogni istanza valida.

Definiamo con il termine di **superchiave** un insieme di attributi $A_1, \ldots, A_n$ tali per cui per ogni altro attributo B nella nelazione R, abbiamo $\{A_1, \ldots, A_n\} \rightarrow B$. In altre parole, una super chiave è un set di uno o più attributi (colonne) che può identificare univocamente una riga in una tabella. Una **chiave** è una super chiave minimale (con un solo attributo) e cioè nessun sottoinsieme di una chiave può essere una super chiave.
L'idea di avere un database normalizzato è utile per la ricerca delle FD (dipendenze funzionali) "cattive", laddove ce ne sia qualcuna. Una volta identificate, basta scomporre la tabella in più sotto-tabelle (di solito collegate tra di loro) fino a che non si hanno più FD "cattive". Una volta concluso, lo schema del database è normalizzato. Le forme normali si distinguono l'una dall'altra in base a ciò che definiscono come FD "cattiva".
Per la Forma Normale Boyce-Codd (Boyce-Codd Normal Form, BCNF) si definisce X → A come una FD "buona" se X è una (super)chiave e "cattiva" altrimenti. La BCNF è una semplice condizione per rimuovere anomalie dalle relazioni: uno schema relazionale R è nella forma BCNF se $\{A_1, \ldots, A_n\} \rightarrow B$ è una FD non banale in R e quindi $\{A_1, \ldots, A_n\}$ è una super chiave per R.
Il problema con la BCNF è che per applicare una FD è necessario ricostruire la relazione originaria su ogni inserimento. La soluzione solitamente risiede in un trade-off tra ridondanza/anomalie nei dati e conservazione della FD.

2 SQL

2.1 Introduzione

La sigla SQL sta per Structured Query Language ed è un linguaggio standard di interrogazione e manipolazione dei dati.
È un linguaggio di programmazione ad alto livello e risulta essere ben ottimizzato.
SQL è un:

- Linguaggio di Definizione dei Dati (Data Definition Language, DDL):
  Definisce lo *schema* della relazione e crea/altera/elimina le tabelle e i loro attributi
- Linguaggio di Manipolazione dei Dati (Data Manipulation Language, DML):
  Inserisce/elimina/modifica le tuple nelle tabelle e interroga una o più tabelle.

Una **relazione** (o una **tabella**) in SQL è un multi-set di tuple che hanno gli attributi specificati dallo schema. Con **multi-set** intendiamo una lista non ordinata (quindi sono permessi più duplicati di una stessa

istanza) e con **attributo** (o colonna) si intende una cella di dati aventi un tipo, presenti in ciascuna tupla della relazione**.** Nel linguaggio SQL standard, un attributo deve essere di tipo atomico. I tipi atomici sono:

- Caratteri (Characters): CHAR(20), VARCHAR(50)
- Numeri (Numbers): INT, BIGINT, SMALLINT, FLOAT
- Altri: MONEY, DATETIME, etc.

Una **tupla** (**riga**) è una singola cella nella tabella che ha gli attributi specificati dallo schema (a volte la si indica con la parola "record").

Lo **schema** di una tabella è il nome della tabella, i suoi attributi e il loro tipo, la chiave è un attributo i cui valori sono unici.

Una **chiave** è un sottoinsieme minimale di attributi che si comportano come unici identificatori per le tuple in una relazione. Una chiave è un'implicita costrizione su quali tuple possono essere incluse nella relazione: così se due tuple sono d'accordo sullo stesso valore della chiave, allora è molto probabile che siano le stesse tuple.

Se alcune informazioni mancano o non sono conosciute, SQL in automatico mostra un valore `NULL`. Per controllare se un valore è `NULL`, viene usato il comando `ISNULL` tipico di SQL. SQL offre la possibilità di forzare una colonna a essere `NOT NULL` o comunque supporta altre limitazioni, come il numero massimo di valori per attributo. Grazie allo schema e alle costrizioni, i database comprendono la semantica dei dati.

2.2 Constraints in SQL

Una costrizione (constraint) è una relazione tra dati a cui un DBMS si deve attenere. I trigger sono eseguiti quando una determinata condizione si verifica. Alcune tipologie di costrizione sono:

- (Foreign) Keys
- Value-Based constraint (costrizione su un certo valore)
- Tuple-Based constraint (relazione tra componenti)
- Assertion (una qualsiasi espressione booleana di SQL)

Quando si crea una tabella, in genere viene anche generata una chiave usando una costrizione di tipo `UNIQUE` (unica) o `PRIMARY KEY` (chiave primaria). Per dichiarare una foreign key, viene messa la parola chiave `REFERENCES` dopo un attributo o come elemento dello schema e l'attributo referenziato deve essere dichiarato (nella tabella a cui appartiene) come `PRIMARY KEY` o `UNIQUE`. Alcuni esempi:

```
--For the single attribute key

CREATE TABLE table_1(
                      attribute_1 CHAR(20) UNIQUE,
                      attribute_2 VARCHAR(20)
                     );

--For the multi-attribute key

CREATE TABLE table_2(
                      attribute_1 CHAR(20),
                      attribute_2 VARCHAR(20),
                      attribute_3 REAL,
                      PRIMARY KEY (attribute_1,attribute_2)
                     );
```

```
--Foreign key reference example

CREATE TABLE table_1(
                      attribute_1 CHAR(20) UNIQUE,
                      attribute_2 VARCHAR(20)
                     );

CREATE TABLE table_2(
                      attribute_3 CHAR(20),
                      attribute_4 VARCHAR(20) REFERENCES table_1(attribute_1),
                      attribute_5 REAL,
                     );

--Foreign key as schema element

CREATE TABLE table_1(
                      attribute_1 CHAR(20) UNIQUE,
                      attribute_2 VARCHAR(20)
                     );

CREATE TABLE table_2(
                      attribute_3 CHAR(20),
                      attribute_4 VARCHAR(20),
                      attribute_5 REAL,
                      FOREIGN KEY(attribute_4) REFERENCES table_1(attribute_1)
                     );
```

Alcune possibili variazioni potrebbero essere causate dalla presenza di una costrizione sulla foreign key:

1. Inserire o aggiornare le operazioni sulla table_2, può introdurre valori non trovati nella table_1;
2. Eliminare o aggiornare le operazioni sulla table_1, causando la disgregazione di alcune tuple della table_2.

Nel primo caso, tali operazioni sono semplicemente respinte. Nel secondo caso, deve essere scelto uno dei seguenti procedimenti:

1. `DEFAULT`: Respinge la modifica nello stesso modo del primo caso;
2. `CASCADE`: Applica gli stessi cambiamenti nella table_2 (cioè in caso di cancellazione (aggiornamento) di una riga della table_1, elimina (aggiorna) tutte le righe corrispondenti della table_2);
3. `SET NULL`: Cambia l'elemento coinvolto con un elemento di tipo NULL.

La sintassi dichiarativa relativa è: `ON [UPDATE, DELETE] [SET NULL CASCADE]` (in genere collocato dopo la dichiarazione della foreign key). Se non viene dichiarata, viene intesa l'opzione di default (respingere). Un esempio:

```
CREATE TABLE table_1(
                      attr_1 CHAR(20) UNIQUE,
                      attr_2 VARCHAR(20)
                     );

CREATE TABLE table_2(
                      attr_3 CHAR(20),
                      attr_4 VARCHAR(20),
                      attr_5 REAL,
```

```
    FOREIGN KEY(attr_4)
    REFERENCES table_1(attr_1)
    ON DELETE SET NULL
    ON UPDATE CASCADE
);
```

Altre costrizioni si hanno con dei controlli Attribute-Based, utilizzati al fine di limitare il valore di un determinato attributo.
In questo caso bisogna aggiungere `CHECK(<condizione>)` alla dichiarazione per l'attributo. Si potrebbe usare il nome dell'attributo nella condizione, ma ogni altra relazione o nome di attributo deve essere in una sotto-query (subquery). I controlli sono performati solo quando viene inserito o aggiornato un valore per quel determinato attributo. Può essere aggiunta come un elemento di relazione-schema (così come la foreign key).

L'ultima costrizione analizzata è l'Assertion, un elemento database-schema. È sintatticamente definita da: `CREATE ASSERTION <nome> CHECK (<condizione>);`
Dove la condizione potrebbe riferirsi a una qualsiasi relazione o a un qualsiasi attributo nello schema del database.

Per esempio, se ci fossero due tabelle, Bars e Drinkers, e si volesse asserire che non possono esserci più bar che bevitori, il codice da inserire sarebbe:

```
CREATE ASSERTION FewBar CHECK (
                                (SELECT COUNT(*) FROM Bars) <=
                                (SELECT COUNT(*) FROM DRINKERS)
                               );
```

In linea di principio, si dovrebbe controllare l'asserzione dopo ogni modificazione nei confronti di ogni relazione del database, ma un sistema intelligente potrebbe notare come solo determinati cambiamenti causerebbero la violazione di una determinata asserzione. Il problema è che il DBMS spesso non è in grado di stabilire quando queste debbano essere controllate. I Trigger permettono all'utente di decidere quando effettuare i controlli su una qualsiasi condizione. Un esempio si ha quando si utilizza una costrizione sulla foreign key; invece che respingere l'inserimento con elementi sconosciuti, un trigger può aggiungere quel determinato nuovo elemento alla tabella originale, impostando gli altri attributi come `NULL`.

2.3 Operazioni di Manipolazione dei Dati SQL

2.3.1 Comandi basilari per una sola tabella

Per proiettare una tabella, possiamo usare una `SELECT FROM`:

```
SELECT <attributes>
FROM <one or more relations>
WHERE [optional]<conditions>
```

Dopo la parola chiave `SELECT`, possiamo usare un `*` per selezionare tutti gli attributi.

Nota bene. I comandi sono "case insensitive" (e cioè insensibili alle maiuscole, non fa differenza) ma i valori sono "case sensitive" (e quindi sensibili, a differenza del caso precedente). Inoltre, quando si utilizzano delle condizioni sulle stringhe bisogna far attenzione a utilizzare una sola virgoletta (single quotes), come 'valore', e non quelle doppie.

Assumiamo di avere una tabella di autori chiamata appunto Authors con il seguente schema:

id	last_name	first_name	DoB	income	genre
INT	CHAR	CHAR	DATETIME	REAL	CHAR

Per gestire i duplicati, può essere messa la parola chiave `DISTINCT` subito dopo la `SELECT`. Associata alla parola chiave WHERE invece possono esserci una o più condizioni (poste usando `AND` oppure `OR`).

Per condizioni sulle stringhe si ha la seguente sintassi: `%` significa "qualsiasi stringa", _ (underscore) significa "qualsiasi carattere" e quando si decide di utilizzarli bisogna ricordarsi di usare una `LIKE` nella clausola `WHERE`. Per ordinarli si deve utilizzare `ORDER BY (DESC) <attributo>`; dove `DESC` serve a ottenere un ordine decrescente, dal momento che l'ordine ottenibile di default è crescente (nel caso, non è sbagliato utilizzare `ASC`). Qui di seguito un esempio usando ciascuna di queste operazioni:

```
--Distinct Genres of Authors whose name start with S and the third letter is a

SELECT DISTINCT genre
FROM Authors
WHERE first_name LIKE 'S_a%'
ORDER BY DESC genre;
```

Per contare il numero di righe viene utilizzato `COUNT(*)` nella clausola `SELECT`. Per contare il numero di valori non nulli in una specifica colonna si usa `COUNT(column1, column2)` nella clausola `SELECT`. Alcune altre operazioni che possono essere fatte nella clausola `SELECT` sono `AVG`, `SUM`, `MIN`, `MAX`.

Queste operazioni sono molto usate insieme alla `GROUP BY`. La `GROUP BY`, a sua volta, risulta utile per aggregare i dati; per calcolare, ad esempio, il reddito medio suddiviso per genere facciamo come segue:

```
SELECT genre, AVG(income)
FROM Authors
GROUP BY genre;
```

Attenzione: nella clausola `SELECT` potremo avere solo gli attributi presenti nella `GROUP BY` o altri attributi purché siano accompagnati da un'operazione di aggregazione, mai da soli. Inoltre, tutte le operazioni menzionate sono da considerare nel caso di una singola tabella.

2.3.2 Comandi basilari su due (o più) tabelle [JOIN/WHERE]

Ora concentriamo la nostra attenzione sul caso di due tabelle. Assumiamo di avere la tabella Authors, la tabella Books, la tabella di join BooksAuthors e una tabella Editions, con il seguente schema:

id	last_name	first_name	DoB	income	genre
INT	CHAR	CHAR	DATETIME	REAL	CHAR

Authors Table

id	title	ISBN
INT	CHAR	BIGINT

Books Table

book_id	author_id
INT	INT

BookAuthors Table

edition_id	book_id	date_of_pubblication	edition_number
INT	INT	INT	INT

Editions Table

Possiamo usare la foreign key per connettere le due tabelle sia nella `FROM`, usando la `JOIN`, che nella `WHERE`.

Qui di seguito i due metodi:

```
--WHERE version to find author of book_id = 1

SELECT first_name, last_name
FROM Authors, BooksAuthors
WHERE BooksAuthors.author_id = Authors.id
      AND book_id = 1;

-- JOIN version to find author of book_id = 1

SELECT first_name, last_name
FROM Authors JOIN BookAuthors
     ON BooksAuthors.author_id = Authors.id
WHERE book_id = 1;

--JOIN version is preferred when only 2 tables are involved
```

Alcuni risultati della `JOIN` potrebbero essere duplicati se la foreign key è spostata/mostrata più di una volta nella tabella. Quindi potrebbe essere utile usare la parola chiave `DISTINCT`. Ad esempio, come si può trovare chi ha scritto un libro il cui ISBN termina con 5 e ha un cognome con esattamente 5 caratteri? Abbiamo bisogno di utilizzare tre tabelle e la versione `WHERE` delle tabelle di collegamento appare come segue:

```
SELECT Books.id, ISBN, Authors.id, last_name, first_name
FROM Books, Authors, BooksAuthors
WHERE Books.id = BooksAuthors.book_id AND
      BooksAuthors.author_id = Authors.id AND
      last_name LIKE '_____' AND
      ISBN LIKE '%5'
```

Laddove fosse fastidioso dover mettere per iscritto frequentemente il nome delle tabelle, possono essere creati degli alias. Ad esempio, rinominiamo `Books` → `b`, `Authors` → `a` e `BooksAuthors` → `ba` e riscriviamo le precedenti interrogazioni alla luce dei nuovi alias:

```
SELECT b.id, ISBN, a.id, last_name, first_name
FROM Books b, Authors a, BooksAuthors ba
WHERE b.id = ba.book_id AND
      ba.author_id = a.id AND
      last_name LIKE '_____' AND
      ISBN LIKE '%5'
```

La semantica `JOIN` tra due tabelle T1 e T2 è un prodotto incrociato tra T1 e T2, seguito da una selezione di righe che soddisfa la clausola `WHERE`, ed eventualmente una proiezione sulle colonne specificate nella clausola `SELECT`.
Assumiamo che un autore non abbia scritto alcun libro, quando eseguiamo un'unione tra `Authors` e `BooksAuthors` quel determinato autore non comparirà. Se vogliamo forzare la sua presenza, dobbiamo usare una `LEFT JOIN` con `Authors` nella `LEFT`. Un esempio della outer `JOIN` (che è un'unione, mentre la inner `JOIN` è un'intersezione) è il seguente:

```
SELECT Authors.id, last_name, first_name, Books.id, ISBN
FROM Books, Authors LEFT JOIN BooksAuthors
     ON Authors.id = BooksAuthors.author_id
WHERE Books.id = BooksAuthors.book_id

--Now the result will also show the authors with Books.id and NULL ISBN value.
```

Potremmo anche rinominare le colonne proiettate usando gli alias. Qui di seguito un esempio dove contiamo il numero di libri scritti da un autore, usando una outer join per includere anche gli autori con zero libri associati:

```
SELECT Authors.id, last_name, first_name,
       COUNT(Books.id) AS Number  --Here is how to rename a column
FROM Books, Authors LEFT JOIN BooksAuthors
     ON Authors.id = BooksAuthors.author_id
WHERE Books.id = BooksAuthors.book_id
GROUP BY Author.id, last_name, first_name; --Here we need all the columns written in the SELECT
```

2.3.3 Operazioni di SET

L'intersezione di due o più tabelle può essere ottenuta usando la clausola **`INTERSECT`**: si comporta come una regolare (in senso matematico) intersezione tra due insiemi (dove gli insiemi corrispondono alle righe). È usata molto raramente, dal momento che viene rimpiazzata spesso da una condizione `WHERE` più complessa (per esempio usando una `AND`). Si mostra un esempio su come trovare gli `Authors` con `Income > 90000` e con il cognome contenente una "o":

```
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000
INTERSECT
SELECT last_name, first_name
FROM Authors
WHERE last_name LIKE '%o%'

--Equivalent WHERE / AND condition

SELECT last_name, first_name
FROM Authors
WHERE Income > 90000 AND last_name LIKE '%o%'
```

Possiamo anche eseguire una **`UNION`** tra due insiemi, ma anche questa è utilizzata molto raramente poiché può essere rimpiazzata da una condizione `WHERE` più complessa, laddove non esistano duplicati. Per eseguire un'unione con duplicati si può utilizzare la `UNION ALL`.
Qui di seguito si riporta un esempio per trovare gli `Authors` con un `Income > 90000` o il nome contenente una "o":

```
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000
```

```
UNION
SELECT last_name, first_name
FROM Authors
WHERE last_name LIKE '%o%'

--If no duplicates equivalent is the WHERE - OR

SELECT last_name, first_name
FROM Authors
WHERE Income > 90000 OR last_name LIKE '%o%'

--Union with duplicates

SELECT last_name, first_name
FROM Authors
WHERE Income > 90000
UNION ALL --Here is the difference
SELECT last_name, first_name
FROM Authors
WHERE last_name LIKE '%o%'
```

Un comando usato spesso nella pratica, relativo alle operazioni con gli insiemi è la **`EXCEPT`**: consiste in una differenza tra due insiemi (tuttavia, può essere rimpiazzato con un'interrogazione nidificata). Esempio:

```
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000
EXCEPT
SELECT last_name, first_name
FROM Authors
WHERE last_name LIKE '%o%'
```

Il codice qui sopra restituirà una lista di autori con `Income > 90000` e il cognome non contenente la lettera "o".

2.3.4 Nested Queries

Come summenzionato, è possibile rimpiazzare alcune operazioni di SET con le query nidificate.

Per rimpiazzare una `INTERSECT` procediamo come segue:

```
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000 AND last_name IN
      (SELECT last_name, first_name
       FROM Authors
       WHERE last_name LIKE '%o%')
```

Per rimpiazzare una `EXCEPT`:

```
SELECT last_name, first_name
FROM Authors
WHERE Income > 90000 AND last_name NOT IN
```

```
(SELECT last_name, first_name
 FROM Authors
 WHERE last_name LIKE '%o%')
```

Le query nidificate hanno molti altri usi, per esempio trovare l'autore con il massimo reddito:

```
SELECT last_name, first_name
FROM Authors A1
WHERE A1.income >= ALL (SELECT A2.income
                        FROM Authors A2)
```

Un altro comando che può essere utilizzato dopo la `WHERE` è `ANY` (invece di `ALL`). In questo caso il risultato includerà ogni riga che soddisfa la condizione per almeno un'altra riga della query nidificata (se viene utilizzata la `ALL`, il risultato è l'insieme di righe che soddisfa la condizione per tutte le righe nella query nidificata).

In SQLite il comando `ALL` è rimpiazzato da `MAX`.