Capitolo 2

# Jaywalking

*Un tecnico di Netscape di cui non faremo il nome una volta passò un puntatore a JavaScript, lo memorizzò in una stringa e poi lo restituì al C… un disastro.*
– Blake Ross

State sviluppando una funzionalità nell'applicazione di bug-tracking per indicare un utente come contatto principale per un prodotto. Il progetto originale consentiva di impostare un solo utente come contatto per ciascun prodotto, ma poi vi è stato chiesto di consentire l'assegnazione di più utenti come contatti per un prodotto dato.

Inizialmente sembrava semplice modificare il database per registrare un elenco di identificatori di account utente separati da virgole, al posto del singolo identificatore utilizzato in precedenza.

Poco dopo il vostro capo vi ha sottoposto un problema: "L'ufficio tecnico ha introdotto nuovo personale nei progetti. Mi hanno detto di poter aggiungere soltanto cinque persone, se provano ad aggiungerne altre, ricevono un errore. Che cosa succede?".

Rispondete: "Sì, si può associare solo un certo numero di persone a un progetto" come se fosse del tutto normale.

Accorgendovi che il capo vuole una spiegazione più precisa: "Be', da cinque a dieci, forse qualcuna in più. Dipende da quanto è vecchio l'account di ogni persona". Ora il capo alza un sopracciglio. Continuate: "Memorizzo gli ID di account per un progetto in un elenco separato da virgole, ma l'elenco degli ID deve stare in una stringa che ha una lunghezza massima. Se

## In questo capitolo

gli ID sono corti, ce ne stanno di più; le persone che hanno creato i primi account hanno ID non più alti di 99, e sono i più corti".
Il capo ha l'aria insoddisfatta. Cominciate a pensare che dovrete fare tardi in ufficio.
I programmatori utilizzano comunemente elenchi separati da virgole per evitare di creare una tabella di riferimento incrociato per una relazione molti a molti. Chiamo questo antipattern *Jaywalking*, perché in inglese il termine indica una persona che attraversa la strada in modo pericoloso, evitando l'incrocio corretto.

## 2.1 Obiettivo: memorizzare attributi a più valori

Quando una colonna di una tabella ha un solo valore, il progetto è immediato: si può scegliere un tipo di dati SQL che rappresenti una singola istanza di tale valore, per esempio un intero, una data o una stringa. Ma come si fa a memorizzare una collezione di valori correlati in una colonna?
Nell'esempio del database di bug-tracking, potremmo associare un prodotto con un contatto utilizzando una colonna di interi nella tabella `Products`. Ogni account potrebbe avere più prodotti, e ciascun prodotto fa riferimento a un solo contatto, perciò abbiamo una relazione *molti a uno* tra prodotti e account.

**Jaywalking/obj/create.sql**

```
CREATE TABLE Products (
   product_id    SERIAL PRIMARY KEY,
   product_name  VARCHAR(1000),
   account_id    BIGINT UNSIGNED,
   -- . . .
   FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', 12);
```

Con la maturazione del progetto, vi accorgete che un prodotto potrebbe avere più contatti. Oltre alla relazione molti a uno, dovete anche supportare una relazione uno a molti da prodotti ad account. È necessario che a una riga della tabella `Products` possa corrispondere più di un contatto.

## 2.2 Antipattern: formato di elenchi separati da virgole

Per ridurre al minimo le modifiche alla struttura del database, decidete di ridefinire la colonna `account_id` come `VARCHAR` in modo da poter elencare più ID di account in tale colonna, separati da virgole.

**Jaywalking/anti/create.sql**

```
CREATE TABLE Products (
   product_id    SERIAL PRIMARY KEY,
   product_name  VARCHAR(1000),
```

```
   account_id    VARCHAR(100), -- elenco separato da virgole
   -- . . .
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34');
```

Sembra la soluzione vincente, perché non avete creato altre tabelle o colonne, avete soltanto modificato il tipo di dati di una sola colonna. Tuttavia, vediamo i problemi di performance e di integrità dei dati che questo progetto comporta.

## Ricerca dei prodotti di un account specifico

Le query sono difficoltose se tutte le chiavi esterne sono combinate in un unico campo. Non potete più utilizzare l'eguaglianza, ma dovete ricorrere a un test di un certo tipo di pattern. Per esempio, MySQL consente di scrivere un codice simile al seguente per trovare tutti i prodotti per l'account `12`:

**Jaywalking/anti/regexp.sql**

```
SELECT * FROM Products WHERE account_id REGEXP '[[:<:]]12[[:>:]]';
```

Le espressioni di pattern-matching possono restituire false corrispondenze e non possono sfruttare gli indici. Poiché la sintassi di pattern-matching varia in base al produttore del database, il codice SQL non è indipendente dal produttore.

## Ricerca degli account per un dato prodotto

Analogamente, è complicato e costoso effettuare il join di un elenco separato da virgole con righe corrispondenti nella tabella referenziata.

**Jaywalking/anti/regexp.sql**

```
SELECT * FROM Products AS p JOIN Accounts AS a
   ON p.account_id REGEXP '[[:<:]]' || a.account_id || '[[:>:]]'
WHERE p.product_id = 123;
```

Effettuare il join di due tabelle utilizzando due espressioni come questa significa perdere ogni possibilità di utilizzare indici. La query deve esaminare entrambe le tabelle, generare un prodotto incrociato e valutare l'espressione regolare per ogni combinazione di righe.

## Query aggregate

Le query aggregate utilizzano funzioni come `COUNT()`, `SUM()` e `AVG()`, che però sono progettate per essere utilizzate su gruppi di righe, non su elenchi separati da virgole.

Dovete ricorrere a trucchi come il seguente:

**Jaywalking/anti/count.sql**

```
SELECT product_id, LENGTH(account_id) - LENGTH(REPLACE(account_id, ',', '')) + 1
   AS contacts_per_product
FROM Products;
```

Trucchi come questo possono essere furbi, ma mai chiari. Le soluzioni di questo tipo richiedono tempo per lo sviluppo e sono ostiche in fase di debugging. Per alcune query aggregate, inoltre, non ci sono proprio trucchi.

## Aggiornamento di account per un prodotto specifico

Potete aggiungere un nuovo ID al termine dell'elenco con la concatenazione di stringhe, ma in questo modo l'elenco risultante potrebbe non essere ordinato.

**Jaywalking/anti/update.sql**

```
UPDATE Products
SET account_id = account_id || ',' || 56
WHERE product_id = 123;
```

Per rimuovere un elemento dall'elenco dovete eseguire due query SQL: una per caricare il vecchio elenco e una seconda per salvare l'elenco aggiornato.

**Jaywalking/anti/remove.php**

```
<?php

$stmt = $pdo->query(
   "SELECT account_id FROM Products WHERE product_id = 123");
$row = $stmt->fetch();
$contact_list = $row['account_id'];

// modifica list nel codice PHP
$value_to_remove = "34";
$contact_list = split(",", $contact_list);
$key_to_remove = array_search($value_to_remove, $contact_list);
unset($contact_list[$key_to_remove]);
$contact_list = join(",", $contact_list);

$stmt = $pdo->prepare(
   "UPDATE Products SET account_id = ?
   WHERE product_id = 123");
$stmt->execute(array($contact_list));
```

È parecchio codice, per rimuovere un elemento da un elenco.

### Validazione di ID di prodotto

Che cosa impedisce a un utente di inserire elementi non validi come *banana*?

**Jaywalking/anti/banana.sql**

```
INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34,banana');
```

Gli utenti troveranno un modo per inserire varianti di ogni tipo, e il database diventerà un ginepraio. Non si tratterà necessariamente di errori di database, ma i dati diventeranno privi di senso.

### Scelta di un carattere separatore

Se memorizzate un elenco di valori stringa invece di interi, alcuni elementi potrebbero contenere il carattere separatore. L'uso di una virgola come separatore tra gli elementi potrebbe risultare ambiguo. Potete scegliere un carattere diverso, ma siete in grado di garantire che questo nuovo separatore non apparirà mai all'interno di un elemento?

### Limitazioni alla lunghezza degli elenchi

Quanti elementi di elenco potete memorizzare in una colonna `VARCHAR(30)`? Dipende dalla lunghezza di ciascun elemento. Se ogni elemento è lungo due caratteri, potete memorizzarne dieci (comprese le virgole), ma se ognuno è lungo sei caratteri, potete memorizzarne soltanto quattro:

**Jaywalking/anti/length.sql**

```
UPDATE Products SET account_id = '10,14,18,22,26,30,34,38,42,46'
WHERE product_id = 123;

UPDATE Products SET account_id = '101418,222630,343842,467790'
WHERE product_id = 123;
```

Come fate a sapere che `VARCHAR(30)` supporta l'elenco più lungo che vi servirà in futuro? Che lunghezza serve? Provate a spiegare le ragioni di questo limite di lunghezza al vostro capo o ai clienti.

## 2.3 Come riconoscere l'antipattern

Se sentite frasi come le seguenti pronunciate dal vostro team di progetto, consideratele un indizio del fatto che viene utilizzato l'antipattern Jaywalking:

- "Qual è il massimo numero di elementi che questo elenco deve supportare?".

  Questa domanda viene sollevata quando state cercando di scegliere la lunghezza massima della colonna `VARCHAR`.

- "Sai come rilevare il limite di una parola in SQL?".

  Se utilizzate espressioni regolari per estrarre parti di una stringa, questo potrebbe essere un indizio del fatto che dovreste memorizzare tali parti separatamente.
- "Quale carattere non apparirà mai in alcun elemento di elenco?".

  Volete utilizzare un carattere separatore non ambiguo, ma dovreste prevedere che qualsiasi carattere possa un giorno o l'altro apparire in un valore dell'elenco.

## 2.4 Impieghi legittimi dell'antipattern

Potreste migliorare le performance per alcuni tipi di query applicando la *denormalizzazione* alla struttura del database. Memorizzare gli elenchi come stringhe separate da virgole è un esempio di questa tecnica.
L'applicazione potrebbe avere la necessità di disporre dei dati in un formato separato da virgole e non di accedere a singoli elementi dell'elenco. Similmente, se l'applicazione riceve da un'altra origine un elenco in formato separato da virgole, e dovete semplicemente memorizzare tutto l'elenco in un database per estrarlo mantenendo esattamente lo stesso formato, non è necessario separare i valori.
Se decidete di ricorrere alla denormalizzazione, usate cautela. Iniziate con una struttura di database normalizzata, che permette una maggiore flessibilità del codice dell'applicazione e aiuta a preservare l'integrità dei dati.

## 2.5 Soluzione: creare una tabella di riferimento incrociato

Anziché memorizzare l'`account_id` nella tabella `Products`, registratelo in una tabella separata, in modo che ogni singolo valore dell'attributo occupi una riga distinta. Questa nuova tabella `Contacts` implementa una relazione *molti a molti* tra `Products` e `Accounts`:

**Jaywalking/soln/create.sql**

```
CREATE TABLE Contacts (
    product_id  BIGINT UNSIGNED NOT NULL,
    account_id  BIGINT UNSIGNED NOT NULL,
    PRIMARY KEY (product_id, account_id),
    FOREIGN KEY (product_id) REFERENCES Products(product_id),
    FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Contacts (product_id, accont_id)
VALUES (123, 12), (123, 34), (345, 23), (567, 12), (567, 34);
```

Quando nella tabella ci sono chiavi esterne che fanno riferimento a due tabelle, si parla di *tabella di riferimento incrociato*, o *tabella di intersezione* (alcuni utilizzano il termine *tabella di join*, *tabella molti a molti*, *tabella di corrispondenza* o altri. Il nome non conta, il concetto è lo stesso). Così si implementa una relazione molti a molti tra le due tabelle referenziate: ogni prodotto può essere associato, attraverso la tabella di riferimento incrociato, a più account, e ogni account può essere associato a più prodotti. Osservate il diagramma entità-relazioni della Figura 2.1.


**Figura 2.1** Diagramma entità-relazioni per la tabella di riferimento incrociato.

Ora vediamo come l'uso di una tabella di riferimento incrociato risolve tutti i problemi descritti nel paragrafo di descrizione dell'antipattern.

## Ricerca di prodotti per account e viceversa

Per cercare gli attributi di tutti i prodotti per un dato account, è più semplice effettuare il join della tabella `Products` con la tabella `Contacts`:

**Jaywalking/soln/join.sql**

```
SELECT p.*
FROM Products AS p JOIN Contacts AS c ON (p.account_id = c.account_id)
WHERE c.account_id = 34;
```

Alcuni non amano le query contenenti un join, perché pensano che offrano prestazioni scarse. Tuttavia, questa query utilizza gli indici molto meglio della soluzione mostrata precedentemente nel paragrafo che descrive l'antipattern.
Anche la query per i dettagli degli account è facile da leggere e da ottimizzare; utilizza gli indici per un join efficiente, anziché ricorrere alle espressioni regolari:

**Jaywalking/soln/join.sql**

```
SELECT a.*
FROM Accounts AS a JOIN Contacts AS c ON (a.account_id = c.account_id)
WHERE c.product_id = 123;
```

## Creare query aggregate

L'esempio che segue restituisce il numero di account per prodotto:

**Jaywalking/soln/group.sql**

```
SELECT product_id, COUNT(*) AS accounts_per_product
FROM Contacts
GROUP BY product_id;
```

Il numero di prodotti per account si ottiene in modo altrettanto semplice:

**Jaywalking/soln/group.sql**

```
SELECT account_id, COUNT(*) AS products_per_account
FROM Contacts
GROUP BY account_id;
```

Si possono ottenere altri report più sofisticati, come il prodotto con il maggior numero di account:

**Jaywalking/soln/group.sql**

```
SELECT c.product_id, c.accounts_per_product
FROM (
    SELECT product_id, COUNT(*) AS accounts_per_product
    FROM Contacts
    GROUP BY product_id
) AS c
HAVING c.accounts_per_product = MAX(c.accounts_per_product)
```

## Aggiornamento dei contatti per uno specifico prodotto

Potete aggiungere o rimuovere elementi dall'elenco inserendo o eliminando righe nella tabella di riferimento incrociato. Ciascun riferimento a un prodotto è memorizzato in una riga separata della tabella `Contacts`, perciò potete aggiungerli o rimuoverli uno alla volta.

**Jaywalking/soln/remove.sql**

```
INSERT INTO Contacts (product_id, account_id) VALUES (456, 34);

DELETE FROM Contacts WHERE product_id = 456 AND account_id = 34;
```

## Validazione di ID di prodotto

Potete utilizzare una chiave esterna per validare gli elementi rispetto a un insieme di valori legittimi contenuti in un'altra tabella. Dichiarate che `Contacts.account_id` fa riferimento ad `Accounts.account_id` e quindi vi affidate al database per imporre l'integrità referenziale. Ora potete essere certi che la tabella di riferimento incrociato contiene soltanto ID di account esistenti.

Potete anche utilizzare tipi di dati SQL per limitare gli elementi inseribili. Per esempio, se gli elementi dell'elenco devono essere valori `INTEGER` o `DATE` validi e dichiarate la colonna utilizzando questi tipi di dati, potete essere certi che tutti gli elementi sono valori leciti di quel tipo (e non voci senza senso come *banana*).

## Scelta di un carattere separatore

Non utilizzate alcun carattere separatore, poiché memorizzate ogni elemento in una riga separata. Non c'è ambiguità se gli elementi contengono virgole o altri caratteri che potrebbero essere usati come separatore.

## Limitazioni alla lunghezza dell'elenco

Poiché ogni elemento si trova in una riga separata della tabella di riferimento incrociato, l'elenco è limitato soltanto dal numero di righe che possono fisicamente esistere in una tabella. Se è appropriato limitare il numero di elementi, dovete applicare il criterio nella vostra applicazione utilizzando il conteggio di elementi, anziché la lunghezza complessiva dell'elenco.

## Altri vantaggi della tabella di riferimento incrociato

Un indice su `Contacts.account_id` permette di migliorare le performance più della corrispondenza di sottostringhe in un elenco separato da virgole. La dichiarazione di una chiave esterna su una colonna crea implicitamente un indice sulla colonna nei database di molti produttori (controllate comunque la documentazione del vostro).
Potete anche creare attributi aggiuntivi per ciascun elemento aggiungendo colonne alla tabella di riferimento incrociato. Per esempio, potreste registrare la data in cui è stato aggiunto un contatto per un prodotto o un attributo per annotare chi sono il contatto principale e quelli secondari. Non potete fare lo stesso in un elenco separato da virgole.

*Memorizzate ciascun valore nella propria colonna e riga.*