

Introduzione ai database e a SQL

Prima di cominciare

Mentre leggete questo capitolo, troverete alcuni frammenti di codice. Se volete eseguire quei frammenti di codice o vedere quali modifiche devono essere apportate al codice per renderlo funzionante su altri database relazionali (RDBMS – *Relational Database Management System*), consultate il repository GitHub che accompagna questo libro (<https://github.com/Neo-Hao/grokking-relational-database-design>). Troverete gli script relativi a questo capitolo nella cartella `chapter_01`; per eseguire gli script seguite le istruzioni riportate nel file `README.md`.

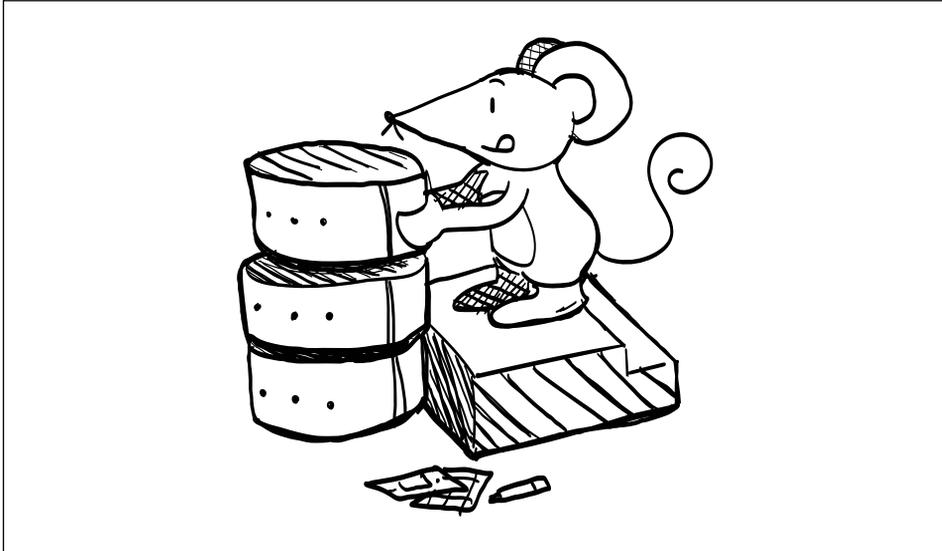
Panoramica

La progettazione del database è una fase critica e spesso trascurata nello sviluppo del software. Quasi ogni applicazione richiede in una certa misura l'archiviazione e la gestione dei suoi dati, ma non tutte le applicazioni impiegano database ben progettati. Se si progetta un database senza conoscerne i principi di progettazione, l'applicazione che lo utilizza potrebbe soffrire di problemi inaspettati, come errori nei dati o query che richiedono troppo tempo e risorse per essere eseguite. Questi problemi possono portare a bug e a una cattiva esperienza utente.

Al contrario, un'efficace progettazione del database può fungere da solida base per uno sviluppo efficace del software.

In questo capitolo

- **Panoramica**
- **La vostra prima query SQL**
- **Le basi delle query SQL**
- **Gestione delle tabelle e dei dati**
- **Riepilogo**



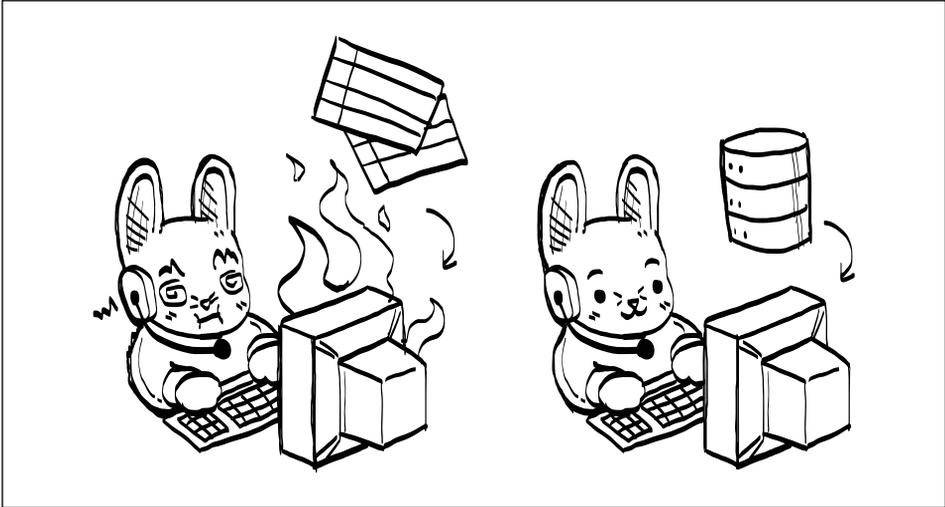
Una progettazione efficace del database assicura che i dati dell'applicazione siano ben organizzati e strutturati, il che a sua volta supporta l'uso di query e manipolazioni dei dati efficienti, che migliorano la solidità delle applicazioni solide e l'esperienza utente. Indipendentemente da dove vi trovate nel vostro percorso di apprendimento della programmazione e dello sviluppo software, è essenziale imparare a progettare i database in modo efficace e possibilmente anche imparare a parlare con persone non esperte di tecnologia senza spaventarli o annoiarli.

Questo libro spiega come progettare un database e non presuppone alcuna conoscenza pregressa, nel campo dei database o della programmazione. Alla fine del libro, avrete buone competenze nel progettare database relazionali da zero. Intendiamo aiutarvi a raggiungere questo obiettivo tramite definizioni e spiegazioni concrete, esempi approfonditi e un apprendimento attivo.

Questo capitolo ha lo scopo di introdurre i database relazionali, definire un set di termini che incontrerete nei prossimi capitoli e trattare le basi del linguaggio SQL (*Structured Query Language*), utilizzato per gestire i dati nei database relazionali; è essenziale che abbiate una certa conoscenza di tale linguaggio per avere una solida comprensione della progettazione di un database.

I database relazionali

C'era una volta una piccola azienda che usava fogli di calcolo di Microsoft Excel per archiviare tutti i dati dei suoi clienti. All'inizio, tutto sembrava funzionare senza intoppi, e l'azienda era in grado di accedere ai dati e aggiornarli quando era necessario. Col passare del tempo, però, l'azienda è cresciuta e ha acquisito più clienti, e così i fogli di calcolo sono diventati sempre più difficili da gestire. Nei dati c'erano duplicazioni e incongruenze, e i fogli di calcolo sono diventati così grandi e ingombranti che ormai ci voleva molto tempo per caricarli e aggiornarli.



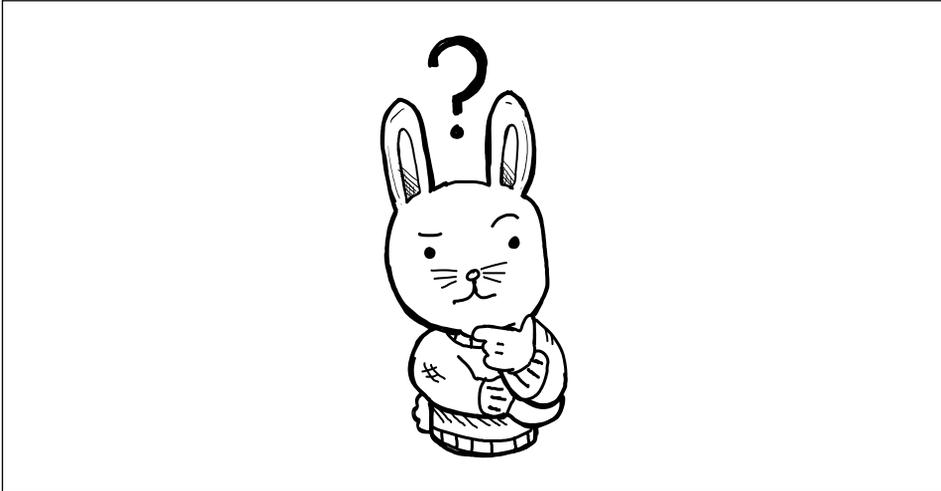
Un giorno, l'azienda ha ricevuto una chiamata da un cliente cui era stato addebitato due volte un certo acquisto. Quando l'azienda ha provato ad accedere ai dati del cliente in un foglio di calcolo per indagare sul problema, ha scoperto che i suoi dati erano danneggiati e non più accessibili. Man mano che sempre più clienti hanno iniziato a segnalare problemi simili, l'azienda ha imparato a sue spese che i fogli di calcolo non sono una buona scelta per archiviare i dati dei clienti. Alla fine, l'azienda ha investito in un database relazionale in grado di gestire la scala dei suoi dati e garantire l'integrità dei record.

Se dovete archiviare una quantità di dati molto piccola con una struttura semplice, un foglio di calcolo può certamente fare al caso vostro; non avete bisogno di un database. Tuttavia, con l'aumento della complessità e del volume dei dati, probabilmente dovrete ripensarci. Quando dovete applicare ai dati il controllo degli accessi, mantenerne la coerenza, l'integrità e la scalabilità e condurre analisi di routine, avete assolutamente bisogno di un database.

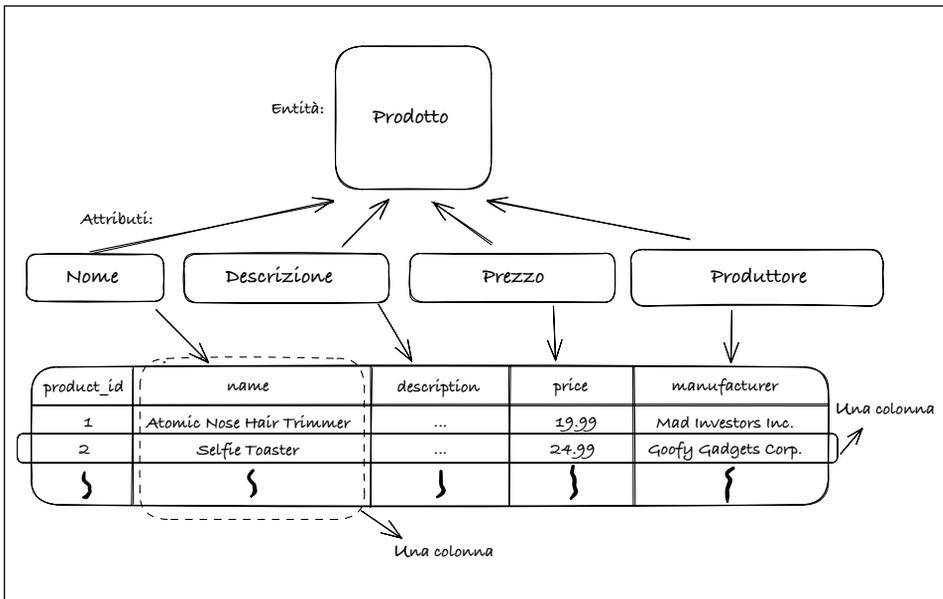
I database relazionali sono stati, e sono tuttora, la tecnologia standard per l'archiviazione e l'accesso ai dati quando sono richieste scalabilità, coerenza e integrità dei dati. Negli ultimi anni, il machine learning e l'intelligenza artificiale hanno contribuito a sostenere e perfino ad aumentare la popolarità dei database relazionali. In questo paragrafo, imparerete alcuni concetti fondamentali dei database relazionali: le tabelle, le entità e i RDBMS.

Le tabelle, le entità e le chiavi primarie

Un *database relazionale* è una raccolta di tabelle contenenti dati. Una *tabella* è come un foglio di calcolo, con cui probabilmente avete più familiarità. Proprio come un foglio di calcolo, i dati di una tabella sono organizzati in righe e colonne. Una tabella può essere utilizzata per rappresentare un'entità o una relazione tra entità, dove ogni riga rappresenta un singolo record di dati di quell'entità e ogni colonna rappresenta un attributo delle entità.



Che cos'è un'entità? Un'entità è un oggetto o un concetto che può essere descritto da più attributi. Supponiamo di gestire un negozio online chiamato *The Sci-Fi Collective* che vende prodotti di fantascienza (come una macchina del tempo che riporta indietro di cinque minuti, nel caso in cui aveste dimenticato le chiavi). I prodotti venduti da The Sci-Fi Collective sono entità, e ognuno di essi può essere descritto da almeno quattro attributi: nome, descrizione, prezzo e produttore. Quando mappiamo i prodotti su una tabella che supporta il negozio online di The Sci-Fi Collective, i quattro attributi diventano quattro colonne e ogni prodotto sarà rappresentato da una riga di questa tabella.



Oltre alle quattro colonne, nella tabella precedente potreste notare che abbiamo aggiunto un'altra colonna, `product_id`. Tutti i valori contenuti nella colonna `product_id` sono univoci e possono essere utilizzati per identificare una singola riga. Chiamiamo `product_id` la *chiave primaria* della tabella `product`. Ogni tabella può avere una sola chiave primaria, che serve per identificare in modo univoco le righe. Potete trovare una discussione molto più approfondita sulle chiavi primarie nel Capitolo 4.

Non è insolito che un foglio di calcolo memorizzi le informazioni di più entità. Potreste chiedervi se sia il caso di fare lo stesso con le tabelle di un database relazionale. Se per The Sci-Fi Collective decidiamo di memorizzare le informazioni di clienti e prodotti in una sola tabella, per esempio, la tabella avrà il seguente aspetto.

<code>product_id</code>	<code>name</code>	<code>price</code>	<code>manufacturer</code>	<code>customer_id</code>	<code>customer_name</code>	<code>customer_email</code>	<code>quantity</code>
1	Atomic Nose ...	19.99	Mad Inventors Inc.	a1	Bob	bob@gmail.com	5
2	Selfie Toaster	24.99	Goofy Gadgets Corp.	b2	Dave	dave@outlook.com	15
3	Cat-Poop Coffee	29.99	Absurd Accessories	a1	Bob	bob@gmail.com	2
...
9	The Infinite ...	9.99	Silly Supplies Co.	j8	John	john@123.net	1
10	The Neuralyzer	33.55	Silly Supplies Co.	p9	Katy	katy@123.net	2

Questa è una tipica tabella mal progettata. Oltre alla ridondanza dei dati, che potete individuare facilmente, una progettazione del genere può causare molti problemi inaspettati. Se le informazioni di un cliente appaiono solo in una riga, per esempio, volendo eliminare un prodotto, dovremo eliminare anche il cliente riportato nella stessa riga. Questo problema è noto come *anomalia di eliminazione*. Considerate un altro esempio: di tanto in tanto dobbiamo inserire in questa tabella un prodotto che nessun cliente ha ancora acquistato, ma la tabella ci chiede di fornire informazioni valide sul cliente ogni volta che aggiungiamo una riga. I requisiti contraddittori ci lasciano in una situazione imbarazzante; non possiamo aggiungere nuovi prodotti. Questo problema è noto come *anomalia di inserimento*.

Come potete vedere, una cattiva progettazione del database può causare problemi che influiscono negativamente sulla qualità del software. Per evitare tali problemi, è importante padroneggiare i principi di base e le best practice della progettazione di database.

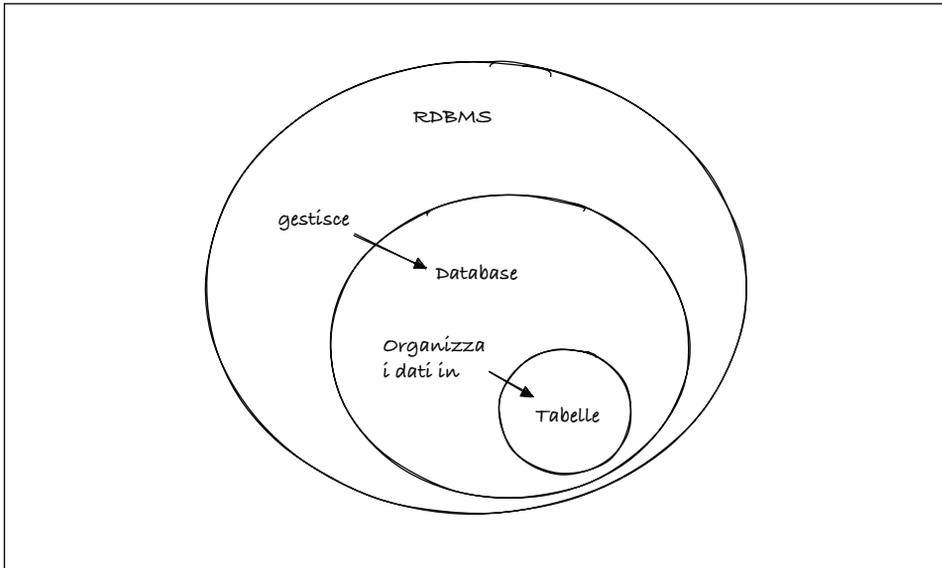
I sistemi di gestione di database relazionali e il linguaggio SQL

I database relazionali e le tabelle si affidano all'aiuto di RDBMS (*Relational Database Management System*) per archiviare e gestire fisicamente i dati. Il primo RDBMS è stato sviluppato negli anni Settanta da Edgar Codd presso IBM.

Un RDBMS è un software che interagisce con l'hardware e il sistema operativo sottostanti per archiviare e gestire fisicamente i dati in database relazionali. Inoltre, un RDBMS fornisce tutti gli strumenti necessari per creare, modificare e interrogare i database, insieme ad altre funzioni importanti come i controlli di sicurezza. Potreste già conoscere, almeno

per nome, alcuni dei database relazionali più usati, come MySQL, MariaDB, PostgreSQL e SQLite. Quando avete bisogno di distribuire un database che avete progettato, dovrete interagire con uno dei database relazionali disponibili sul mercato.

Uno degli strumenti più degni di nota supportati da quasi tutti gli RDBMS è SQL, un linguaggio di programmazione che permette di creare, modificare e interrogare i dati archiviati nelle tabelle di un RDBMS. Sebbene diversi fornitori di RDBMS possano implementare le proprie varianti ed estensioni, SQL è stato standardizzato nel corso degli anni.



Di conseguenza, la coerenza del linguaggio SQL tra i vari RDBMS è elevata, e le varianti non hanno particolare rilevanza nel contesto di questo libro.

Poiché questo libro riguarda principalmente la progettazione di database, SQL potrebbe sembrare meno importante. La progettazione di database non richiede necessariamente l'uso del linguaggio SQL. Alcuni RDBMS sono dotati di strumenti grafici per generare, dietro le quinte, script SQL che creano automaticamente database e tabelle in base alle indicazioni progettuali fornite. Ma sapere come funziona il linguaggio SQL può semplificare l'apprendimento della progettazione di database, soprattutto quando si devono considerare problemi strutturali o progettuali come l'integrità dei dati, l'ottimizzazione e la scalabilità. Dopo tutto, SQL è un linguaggio standard utilizzato dalla maggior parte dei database relazionali, quindi conoscere SQL vi consentirà di fare meno affidamento sugli strumenti grafici e di lavorare con diversi tipi di RDBMS. Tratteremo le basi del linguaggio SQL in questo capitolo e nel Capitolo 2.

La vostra prima query SQL

In questo paragrafo, imparerete a usare il linguaggio SQL eseguendo la vostra prima query. Utilizzeremo l'esempio che abbiamo visto nel paragrafo precedente, il database di

The Sci-Fi Collective (perché a chi non piacciono le cose di fantascienza?). Il database contiene molte tabelle, ma per ora ci concentreremo sulla tabella `product`, che si presenta come segue:

<code>product_id</code>	<code>name</code>	<code>description</code>	<code>price</code>	<code>manufacturer</code>
1	Atomic Nose Hair Trimmer	...	19.99	Mad Inventors Inc.
2	Selfie Toaster	...	24.99	Goofy Gadgets Corp.
3	Cat-Poop Coffee	...	29.99	Absurd Accessories
...
9	The Infinite Improbability Generator	...	9.99	Silly Supplies Co.
10	The Neuralyzer	...	33.55	Silly Supplies Co.

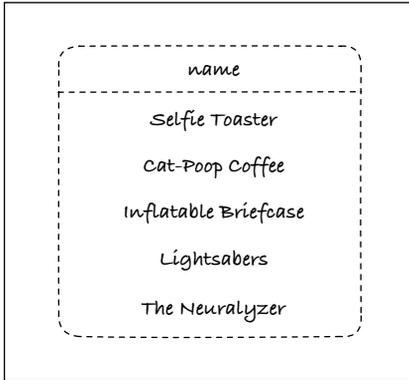
Per prima cosa, caricheremo uno script SQL già pronto all'uso per generare un database e questa tabella. Abbiamo preparato gli script SQL che generano questa tabella con i relativi dati, che trovate nel repository GitHub (<https://github.com/Neo-Hao/grokking-relational-database-design>). Per eseguire lo script per il vostro database relazionale o strumento preferito potete seguire le istruzioni contenute nel file `README.md` nella cartella `chapter_01`. L'approccio più semplice consiste nell'usare *SQLite Online*, come segue.

1. Clonate o scaricate il repository GitHub (<https://github.com/Neo-Hao/grokking-relational-database-design>).
2. Andate su *SQLite Online* (<https://sqliteonline.com>).
3. Selezionate il vostro database relazionale nella barra laterale sinistra e fate clic sul pulsante per connettervi.
4. Fate clic su *Import* e caricate lo script corrispondente al database relazionale scelto (per esempio `mysql_db.sql` dal repository GitHub scaricato o clonato per MariaDB).
5. Fate clic su OK.

Dopodiché, sarete pronti per interrogare la tabella `product`. Per esempio, potete digitare (per intero) la seguente query nell'editor di codice di *SQLite Online* e poi fare clic su *Run*:

```
SELECT name
FROM product
WHERE price > 20;
```

Che cosa fa questa query? La parte `price > 20` potrebbe essere un indizio. La query recupera i nomi dei prodotti il cui prezzo è maggiore di 20. Sappiamo che nella tabella `product` ci sono dieci righe che rappresentano altrettanti prodotti e che cinque prodotti (come Atomic Nose Hair Trimmer) vengono venduti a un prezzo inferiore a 20, quindi verranno restituiti i nomi degli altri cinque prodotti. Il risultato dovrebbe essere simile a questo:



Potreste notare che questa query SQL ha molte somiglianze dirette con l'inglese. Il motivo è che SQL è un linguaggio speciale. La maggior parte dei linguaggi di programmazione ha una natura *imperativa*. La programmazione con un linguaggio imperativo, come Java o Python, richiede di specificare sia che cosa occorre fare sia come farlo. SQL, al contrario, è *dichiarativo*, il che significa che dovete specificare solo quello che volete ottenere. I passaggi richiesti per eseguire l'attività sono tutti a carico del linguaggio SQL. Specificare che cosa serve fare (e non come riuscirci) è più naturale per noi esseri umani, ed è per questo che SQL somiglia così tanto all'inglese colloquiale.

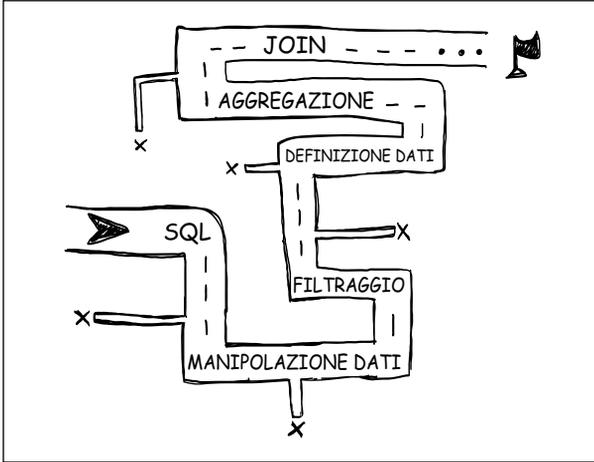
Per la precisione, SQL è come l'inglese, ma senza orpelli. Non c'è molta libertà nel comporre una query SQL. Dovete usare un preciso set di clausole SQL (chiamate *istruzioni*) e seguire alcune regole. Nella vostra prima query, avete usato le seguenti tre clausole.

- La clausola `SELECT` consente di specificare le colonne che volete estrarre da una tabella. Nella prima query, è stata richiesta solo la colonna `name`; pertanto, l'istruzione `SELECT` era `SELECT name`.
- La clausola `FROM` specifica la fonte da cui volete recuperare i dati, da una o più tabelle. Nella vostra prima query, avete chiesto solo dati tratti dalla tabella `product`; quindi, la clausola `FROM` era `FROM product`.
- La clausola `WHERE` consente di specificare le condizioni per filtrare i dati recuperati dalla clausola `SELECT`. Nella prima query, cercavamo il nome (`name`) dei soli prodotti i cui prezzi erano superiori a 20; pertanto, la query era `SELECT name FROM product WHERE price > 20;`.

Al termine di una query SQL, occorre sempre usare un punto e virgola (;). Il punto e virgola dice al database relazionale che questa è la fine della query SQL e che tutto ciò che seguirà apparterrà a una nuova query.

Le basi delle query SQL

Il nostro approccio preferito per imparare a usare il linguaggio SQL consiste nel presentare le clausole più importanti e imparare il resto solo quando è necessario. Sebbene il linguaggio SQL abbia molte clausole, non tutte sono ugualmente importanti. Quelle più importanti possono aiutarvi a creare una solida base, ma anche a costruire una mappa mentale che può guidare il vostro successivo apprendimento.



Pertanto, invece di provare a trattare ogni singola clausola SQL, esamineremo insieme solo quelle che sono essenziali o critiche per l'apprendimento. In questo capitolo, esamineremo solo clausole importanti che potete usare per interrogare una singola tabella.

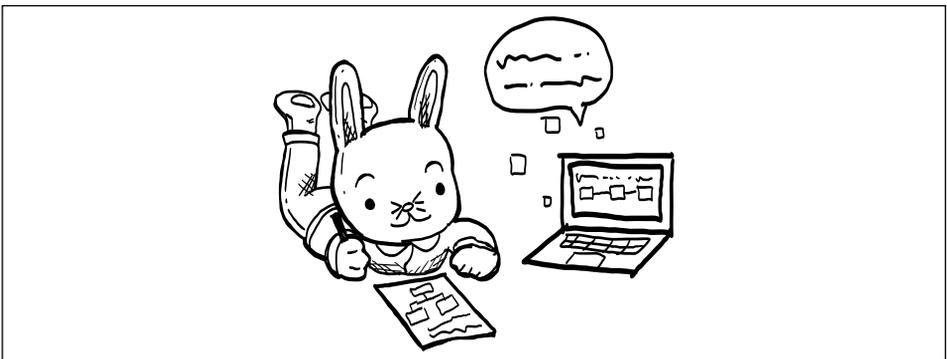
Filtraggio

Il filtraggio è un'attività comune nell'estrazione di dati da un database. Ogni volta che avete bisogno di un sottoinsieme dei dati, solo quelli che soddisfano determinati criteri, avete bisogno della clausola `WHERE` per filtrare i dati.

Dalla vostra prima query SQL, sapete che la clausola `WHERE` è seguita dai criteri che volete usare per filtrare i dati. La seguente query, per esempio, estrae dalla tabella `product` il nome e la descrizione dei prodotti il cui prezzo è inferiore a 30:

```
SELECT name, description
FROM product
WHERE price < 30;
```

Se volete estrarre dati da più di una colonna, potete elencare tutte le colonne che volete dopo la parola chiave `SELECT`, separate da una virgola.



E se volessimo estrarre solo i prodotti di un certo produttore, come Mad Inventors Inc.? Possiamo farlo tramite la seguente query:

```
SELECT name
FROM product
WHERE manufacturer = 'Mad Inventors Inc.';
```

Questa query produce il seguente risultato:

<i>name</i>
<i>Atomic Nose Hair Trimmer</i>
<i>The Mind Probe</i>
<i>Lightsabers</i>

Nella query precedente, l'operatore che verifica l'uguaglianza è un singolo segno di uguale (=). Inoltre, potreste notare che il nome del produttore è racchiuso tra virgolette singole (' '), per indicare che si tratta di una stringa. SQL impiega diversi tipi di dati? Sì. I dati SQL possono essere suddivisi in sei categorie:

- dati di tipo numerico (come INT);
- dati di tipo stringa (per esempio TEXT);
- dati di tipo data o ora (per esempio DATE);
- dati di tipo stringa di caratteri Unicode (come VARCHAR);
- dati di tipo binario (come BINARY);
- dati vari (come XML).

Nella tabella `product`, i dati della colonna `manufacturer` sono di tipo stringa. Al contrario, la colonna `price` è numerica.

Ora che sapete come filtrare i dati numerici e di tipo stringa, potete creare un filtro che combina i due criteri usando gli operatori logici. AND e OR sono i due operatori logici più utilizzati. L'operatore AND ha lo stesso significato che ha in inglese e lo stesso si può dire di OR. Possiamo combinare con AND due criteri nel seguente modo:

```
SELECT *
FROM product
WHERE price < 30 AND manufacturer = 'Mad Inventors Inc.';
```

Questa query produce il seguente risultato:

product_id	name	description	price	manufacturer
1	Atomic Nose Hair Trimmer	...	19.99	Mad Inventors Inc.
6	The Mind Probe	...	19.99	Mad Inventors Inc.
7	Lightsabers	...	25.00	Mad Inventors Inc.

A differenza delle precedenti, questa query estrae ogni colonna dalla tabella `product`. L'asterisco (*) che segue la parola chiave `SELECT` indica "tutte le colonne". La combinazione dei due criteri di filtraggio recupera tutte le colonne ma dei soli prodotti fabbricati da Mad Inventors Inc. e anche con un prezzo inferiore a 30.

Aggregazione

L'aggregazione, un'attività importante in SQL, comporta l'esecuzione di calcoli su un set di righe per produrre un singolo risultato. Aggregando i dati, potete ottenere informazioni sulle tendenze insite nei dati, che potrebbero non essere visibili osservando i dati a livello del singolo record. Le funzioni di aggregazione utilizzate più di frequente sono le seguenti.

- `COUNT()`: conta il numero di righe.
- `SUM()`: calcola la somma dei valori in una colonna numerica.
- `AVG()`: calcola la media dei valori in una colonna numerica.
- `MAX()`: trova il valore massimo in una colonna.
- `MIN()`: trova il valore minimo in una colonna.

Quando formuliamo una query SQL che implica un'aggregazione, dobbiamo posizionare la funzione di aggregazione nell'istruzione `SELECT`. Per esempio, possiamo contare il numero di righe presenti nella tabella `product` nel seguente modo:

```
SELECT COUNT(*) FROM product;
```

Questa query produce il seguente risultato:

COUNT(*)
10

Nella figura, potrete notare che la colonna ha lo stesso nome della funzione di aggregazione. Se usate un database relazionale diverso da MariaDB, il nome della colonna potrebbe essere `COUNT()` o qualcos'altro. Se non vi piace questo nome, potete fornirne uno più leggibile, usando un alias tramite la clausola `AS`. Potete calcolare il prezzo medio di tutti i prodotti venduti nel negozio e usare `avg_price` per il nome della colonna, come in questo esempio:

```
SELECT AVG(price) AS avg_price
FROM product
WHERE manufacturer = 'Mad Inventors Inc.';
```

Questa query produce il seguente risultato, in cui il nome della colonna è `avg_price`, e l'unico valore restituito è la media di tutti i prezzi dei prodotti presenti nella tabella:

<code>avg_price</code>
21.66

In entrambi gli esempi, avete applicato le funzioni di aggregazione a tutte le righe di una tabella. Tuttavia, potete applicare le funzioni di aggregazione anche a determinati gruppi di righe di una tabella. A volte, occorre raggruppare i dati per una o più colonne e analizzare i dati raggruppati. Per raggruppare i dati si usa la clausola `GROUP BY`, che è comunemente usata in combinazione con le funzioni di aggregazione. La clausola `GROUP BY` è sempre seguita da uno o più nomi di attributi, separati da una virgola. Per esempio, potete contare il numero di prodotti per produttore nel seguente modo:

```
SELECT COUNT(*) AS product_count, manufacturer
FROM product
GROUP BY manufacturer;
```

Questa query produce il seguente risultato, anche se l'ordine delle righe può variare:

<code>product_count</code>	<code>manufacturer</code>
2	Absurd Accessories
1	Goofy Gadgets Corp.
3	Mad Inventors Inc.
3	Silly Supplies Co.
1	Wacky Wares Ltd.

Come altro esempio, potete calcolare il prezzo medio dei prodotti per produttore:

```
SELECT AVG(price) AS avg_price, manufacturer
FROM product
GROUP BY manufacturer;
```

Questa query produce il seguente risultato.

<i>avg_price</i>	<i>manufacturer</i>
22.5449...	Absurd Accessories
24.99	Goofy Gadgets Corp.
21.659...	Mad Inventors Inc.
17.843...	Silly Supplies Co.
39.99	Wacky Wares Ltd.

Quando si utilizzano le funzioni di aggregazione con la clausola `GROUP BY`, è necessario includere nell'istruzione `SELECT` anche gli attributi che seguono la clausola `GROUP BY`, altrimenti, i risultati potrebbero non avere molto senso. La seguente query raggruppa i dati in base alla colonna `manufacturer`, ma non la include nell'istruzione `SELECT`:

```
--commento: produrrà un output di difficile interpretazione
SELECT COUNT(*) AS product_count
FROM product
GROUP BY manufacturer;
```

Il risultato non sarà molto utile, perché vedrete solo una colonna di numeri e non avrete idea di cosa essi rappresentino:

<i>product_count</i>
2
1
3
3
1

Come altro esempio, la seguente query calcola il prezzo medio dei prodotti per produttore, ma non include la colonna `manufacturer` nell'istruzione `SELECT`:

```
--commento: produrrà un output di difficile interpretazione
SELECT AVG(price) AS avg_price
FROM product
GROUP BY manufacturer;
```

Come nell'esempio precedente, il risultato è difficile da interpretare, perché vedete solo una colonna di numeri decimali e non avete idea di cosa essi rappresentino:

<i>avg_price</i>
22.5449...
24.99
21.659...
17.843...
39.99

Ancora più importante, dovrete escludere dall'istruzione `SELECT` tutte le colonne che non sono presenti nella clausola `GROUP BY`, a meno che non siano utilizzate nelle funzioni di aggregazione. La seguente query tenta di contare il numero di prodotti per produttore, ma la colonna `name` nell'istruzione `SELECT` non è né presente nella clausola `GROUP BY` né è utilizzata in una funzione di aggregazione:

```
-- Commento - Porterà a un errore o produrrà un risultato fuorviante
SELECT COUNT(*) AS product_count, manufacturer, name
FROM product
GROUP BY manufacturer;
```

Una query come questa causerà errori o produrrà un risultato fuorviante, a seconda del database relazionale che utilizzate. PostgreSQL, per esempio, si lamenterà così:

```
ERROR: column "product.name" must appear in the GROUP BY clause
or be used in an aggregate function
```

SQLite, invece, produrrà un risultato fuorviante ma senza lamentarsi:

<i>product_number</i>	<i>manufacturer</i>	<i>name</i>
2	Absurd Accessories	Cat-Poop Coffee
1	Goofy Gadgets Corp.	Selfie Toaster
3	Mad Inventors Inc.	Atomic Nose Hair Trimmer
3	Silly Supplies Co.	Unicorn Hair Polish
1	Wacky Wares Ltd.	Inflatable Briefcase



Se controllate tutti i dieci prodotti presenti nella tabella `product`, vedrete che c'è solo un Atomic Nose Hair Trimmer e non tre. Ma poiché la query non sa come gestire la colonna `name`, mostra semplicemente il nome presente nella prima riga che incontra nel gruppo. Un altro esempio: la seguente query tenta di calcolare il prezzo medio dei prodotti per produttore, ma la colonna `product_id` dell'istruzione `SELECT` non è presente nella clausola `GROUP BY`:

```
-- Commento - Porterà a un errore o produrrà un risultato fuorviante
SELECT product_id, AVG(price) AS avg_price, manufacturer
FROM product
GROUP BY manufacturer;
```

A seconda del database relazionale che utilizzate, potreste ottenere un errore o un risultato fuorviante. PostgreSQL, per esempio, si lamenterà così:

```
ERROR: column "product.product_id" must appear in the GROUP BY clause
or be used in an aggregate function
```

SQLite produrrà un risultato fuorviante, ma senza lamentarsi:

<code>product_id</code>	<code>avg_price</code>	<code>manufacturer</code>
3	22.5449...	Absurd Accessories
2	24.99	Goofy Gadgets Corp.
1	21.659...	Mad Inventors Inc.
5	17.843...	Silly Supplies Co.
4	39.99	Wacky Wares Ltd.



Se controllate la terza riga della tabella `product`, vedrete che il suo prezzo è 29,99, non 22,5449... Il risultato aggregato precedente è ovviamente sbagliato. Poiché questa query non sa come gestire la colonna `product_id`, mostra semplicemente il primo valore `product_id` che incontra per il gruppo di produttori.

In sintesi, quando si utilizzano le funzioni di aggregazione con la clausola `GROUP BY`, bisogna fare attenzione a quali attributi includere nell'istruzione `SELECT`. L'istruzione `SELECT` dovrebbe contenere solo gli attributi (non aggregati) che compaiono nella clausola `GROUP BY`. Sono attributi non aggregati quegli attributi che non sono coinvolti nell'aggregazione.

Gestione delle tabelle e dei dati

Finora, avete applicato istruzioni SQL a una tabella che vi abbiamo fornito, ma a volte avete anche bisogno di creare le vostre tabelle e poi di gestire tali tabelle e i relativi dati.

La gestione delle tabelle e dei dati è un'attività importante in SQL. I comandi SQL dedicati a tale attività appartengono alla “categoria” DDL (*Data Definition Language*), ovvero formano una sorta di “linguaggio per la definizione dei dati”. Al contrario, le clausole e le istruzioni SQL che avete visto nei paragrafi precedenti sono note cumulativamente come DML (*Data Manipulation Language*), ovvero “linguaggio per la manipolazione dei dati”. Conoscere un po' di DDL è particolarmente utile per la progettazione di database. In questo paragrafo, imparerete tre attività comuni di gestione delle tabelle: la creazione, la modifica e l'eliminazione di una tabella. Imparerete anche ad aggiungere dati a una nuova tabella.



Creazione di tabelle e aggiunta di dati

Ora imparerete a creare una tabella e ad aggiungervi dati dagli script che accompagnano questo capitolo. Gli script (come `mysql_db.sql` per MySQL o MariaDB) hanno lo scopo di creare la tabella `product` per il database che supporta lo shop online The Sci-Fi Collective e poi di popolarla con un set di dati campione. La tabella `product` viene creata col seguente comando:

```
CREATE TABLE product(
  product_id INT PRIMARY KEY,
  name TEXT NOT NULL,
  description TEXT NOT NULL,
  price DECIMAL(5, 2) NOT NULL,
  manufacturer TEXT NOT NULL
);
```

Dobbiamo rispondere a due domande relative al comando che crea la tabella `product`.

- Qual è la sintassi generale per creare una tabella?
- A che cosa servono le diverse parole chiave di questa query?

Quando le tabelle si arrabbiano per le duplicazioni

Se avete seguito e importato lo script SQL indicato nei paragrafi precedenti, avrete notato alcuni reclami da parte del linguaggio SQL quando provate a eseguire il seguente comando `CREATE TABLE`. Perché? Perché, quando avete importato lo script SQL per la vostra prima query SQL, avete eseguito quello stesso comando che ha creato una tabella denominata `product`. Lo stesso RDBMS non può avere due tabelle con lo stesso nome. Lo stesso vale per i comandi di inserimento di dati, che vedremo più avanti. Una tabella non può avere due righe identiche, in particolare mai due righe con la stessa chiave primaria.

Se volete eseguire i comandi `CREATE TABLE` e di inserimento dati trattati in questo paragrafo dopo aver importato gli script preparati, potete procedere come segue.

- Se utilizzate SQLite Online, potete facilmente reimpostare tutto aggiornando la scheda del browser.
- Se utilizzate un database relazionale in esecuzione sul sistema locale o su un server, dovrete prima eliminare la tabella corrispondente. Più avanti parleremo anche dell'eliminazione e della modifica delle tabelle.

Qual è la sintassi generale per creare una tabella? Per creare una tabella, avete bisogno del comando `CREATE TABLE`. La sintassi del comando `CREATE TABLE` è la seguente:

```
CREATE TABLE nome_tabella (
  nome_colonna1 tipo_di_dati [parametri_opzionali],
  nome_colonna2 tipo_di_dati [parametri_opzionali],
  ...
  nome_colonnaN tipo_di_dati [parametri_opzionali],
  PRIMARY KEY (nome_colonnaX)
);
```

La definizione della chiave primaria può anche essere inserita nella definizione di una colonna, come avete visto nell'esempio della tabella `product`. A differenza dei parametri opzionali, la chiave primaria è obbligatoria in ogni tabella.

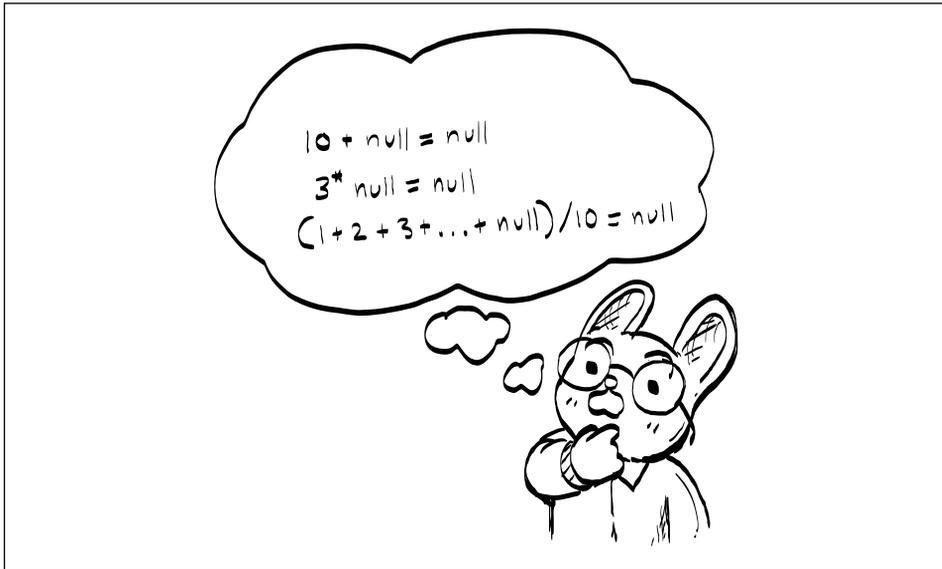
Che cosa fanno le diverse parole chiave in questa query? Per prima cosa, abbiamo specificato il nome della tabella, `product`, e definito cinque colonne:

- `product_id` con tipo di dati numerico (`INT`);
- `name`, con tipo di dati per stringhe (`TEXT`);
- `description`, con tipo di dati per stringhe (`TEXT`);
- `price`, con tipo di dati numerico (`DECIMAL`);
- `manufacturer`, con tipo di dati per stringhe (`TEXT`).

`INT` indica un intero e `TEXT` indica una stringa. L'unico tipo di dati che vale la pena di spiegare qui è probabilmente `DECIMAL`. `DECIMAL`, come tipo di dati numerico, accetta due parametri. Il primo definisce il numero totale di cifre e il secondo il numero di cifre a destra del punto decimale. Usiamo `DECIMAL(5,2)`, per esempio, per l'attributo `price` per consentire l'uso di cinque cifre totali, con due cifre a destra del punto decimale.

Oltre ai tipi di dati, potreste notare che abbiamo specificato ogni attributo come `NOT NULL`. In SQL, un valore `NULL` rappresenta un valore sconosciuto, un po' come quando si cerca di

ricordare il nome di qualcuno che è lì sulla punta della lingua. Consentire l'uso di valori NULL per gli attributi può portare SQL ad avere comportamenti inaspettati. Sommando 10 a un valore che è a NULL, per esempio, ottenete ancora un valore NULL; la somma di un valore sconosciuto e 10 è essa stessa sconosciuta. Quando eseguite calcoli su valori NULL, tutti i risultati finiscono per essere NULL.



Infine, è stata utilizzata la definizione PRIMARY KEY per specificare quale attributo vogliamo utilizzare come chiave primaria per questa tabella. Il nome dell'attributo deve essere specificato tra parentesi dopo PRIMARY KEY.

Ora che la tabella product è stata creata in un database, è già pronta per l'aggiunta di dati. Per aggiungere dati a una tabella, avete bisogno del comando INSERT INTO che vi consente di inserire una o più righe di dati in una tabella. Ecco la sua sintassi di base:

```
INSERT INTO nome_tabella (colonna1, colonna2, ...)
VALUES (valore1, valore2, ...);
```

Nello stesso script, potete trovare un esempio di aggiunta di dati alla tabella product. Per inserire una singola riga nella tabella potete procedere come segue:

```
INSERT INTO product (product_id, name, description, price, manufacturer)
VALUES (
  1,
  'Atomic Nose Hair Trimmer',
  'Trim your nose hairs... of an atomic clock!',
  19,99,
  'Mad Inventors Inc.'
);
```

Oppure potete inserire più righe di dati nella tabella.

```
INSERT INTO product (product_id, name, description, price, manufacturer)
VALUES
(
  2,
  'Selfie Toaster',
  'Get your face on... with our selfie toaster',
  24.99,
  'Goofy Gadgets Corp.'
),
(
  3,
  'Cat-Poop Coffee',
  'The only coffee... the finest cat poop ...',
  29.99,
  'Absurd Accessories'
);
```

Modifica e rimozione di tabelle

Di tanto in tanto potrebbe essere necessario modificare o eliminare una tabella esistente perché a volte è necessario riorganizzare le cose nei dati.

Esistono molti modi per modificare una tabella: potete aggiungere una nuova colonna, modificare il tipo di dati di una colonna o rinominare l'intera tabella. Per queste attività potete usare il comando `ALTER TABLE`. Se per esempio volete aggiungere alla tabella `product` una colonna che rappresenti i numeri di serie, potete usare la seguente query:

```
ALTER TABLE product
ADD serial_number INT;
```

Eseguendo la query precedente, aggiungerete a questa tabella una nuova colonna denominata `serial_number` e con tipo di dati `INT`. Se poi vi rendete conto che `INT` non è il tipo di dati migliore per i numeri di serie, potete aggiornare il suo tipo in una stringa tramite la seguente query:

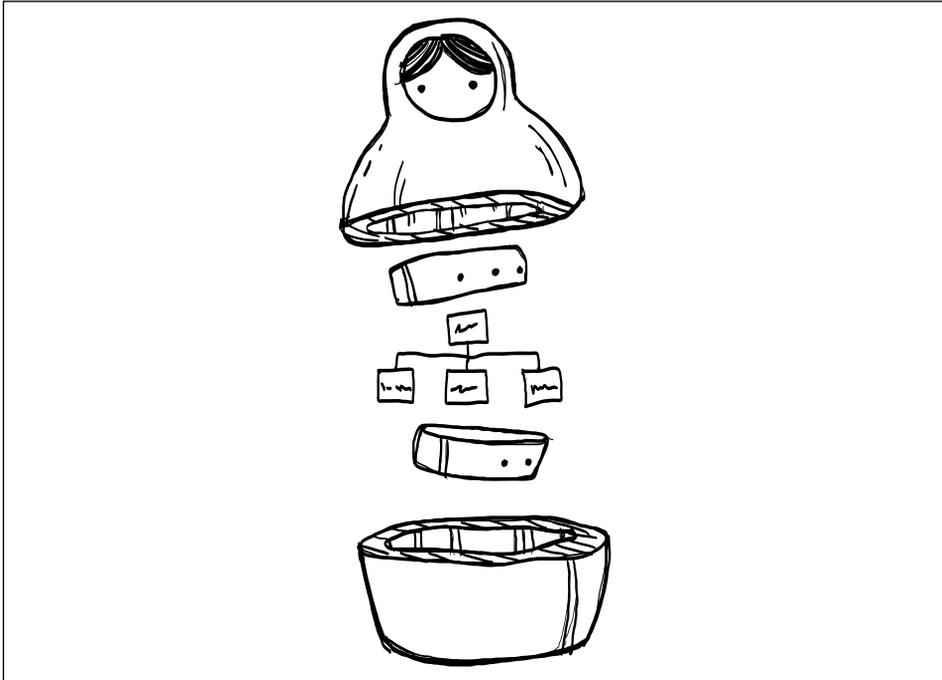
```
-- Commento - SQLite non supporta la modifica diretta del tipo di dati di una colonna
ALTER TABLE product
ALTER COLUMN serial_number TEXT;
```

Mentre esistono molti modi per modificare una tabella, esiste un solo modo per eliminarla. Per farlo, si usa il comando `DROP TABLE` seguito dal nome della tabella da eliminare. Per eliminare la tabella `product`, per esempio, potete utilizzare la seguente query:

```
DROP TABLE product;
```

Occorre prestare attenzione quando si utilizza il comando `DROP TABLE`, perché eliminerà definitivamente la tabella e tutti i suoi dati.

Questo paragrafo non aveva lo scopo di essere un elenco esaustivo di tutti i comandi impiegabili per modificare o eliminare una tabella. Per saperne di più, consultate un manuale su SQL o sul vostro database relazionale. Detto questo, avete compiuto un passo importante nell'impadronirvi dei database e nella progettazione di database. I concetti che avete acquisito in questo capitolo sosterranno il vostro apprendimento nel resto del libro.



Riepilogo

- Un database relazionale è una raccolta di tabelle che memorizzano dati.
- Una tabella si utilizza per rappresentare un'entità o una relazione tra entità in un database.
- Un'entità è un oggetto o un concetto che può essere descritto da più attributi.
- Un RDBMS (*Relational Database Management System*) è un software che interagisce con l'hardware e il sistema operativo sottostanti per archiviare e gestire fisicamente i dati conservati nei database relazionali.
- Per filtrare i dati è necessario impiegare almeno tre clausole SQL: `SELECT`, `FROM` e `WHERE`.
- Spesso, con la clausola `GROUP BY` vengono usate apposite funzioni di aggregazione dei dati.
- I comandi SQL utilizzati per gestire le tabelle sono noti collettivamente col nome DDL (*Data Definition Language*). La gestione delle tabelle, in genere, prevede l'uso di tre comandi: `CREATE TABLE`, `ALTER TABLE` e `DROP TABLE`.
- Per inserire una o più righe di dati in una tabella si usa l'istruzione `INSERT TO ... VALUE`.

Correlazioni fra tabelle e altro codice SQL

Prima di cominciare

Per comprendere il codice SQL presentato in questo capitolo, è necessario sapere che cosa sono i database e le tabelle, nonché le basi delle query SQL. Se avete dubbi, leggete prima il Capitolo 1.

Mentre leggete questo capitolo, troverete altri frammenti di codice. Se volete eseguire quei frammenti di codice o trovare le loro varianti per altri database relazionali, consultate il repository GitHub che accompagna questo libro (<https://github.com/Neo-Hao/grokking-relational-database-design>). Potete trovare gli script relativi a questo capitolo nella cartella `chapter_02`; per eseguire gli script seguite le istruzioni riportate nel file `README.md`.

L'approccio più semplice consiste nell'utilizzare SQLite Online come segue.

1. Clonate o scaricate il repository GitHub (<https://github.com/Neo-Hao/grokking-relational-database-design>).
2. Individuate gli script nella cartella `chapter_02`.
3. Visitate *SQLite Online* (<https://sqliteonline.com>). Scegliete il vostro database relazionale nella barra laterale sinistra e fate clic su *Click to Connect*.
4. Fate clic su *Import* per caricare lo script corrispondente al database relazionale scelto (per esempio `mysql_db.sql` dal repository GitHub scaricato o clonato per MariaDB).
5. Fate clic su OK.

In questo capitolo

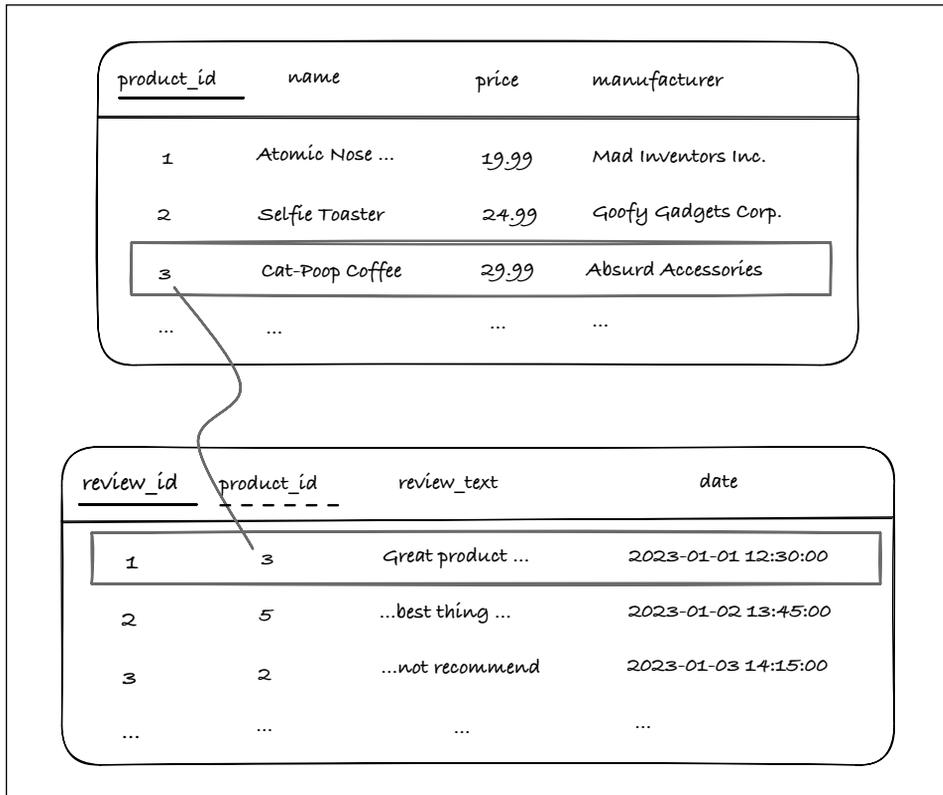
- **Panoramica**
- **Tabelle correlate**
- **Di nuovo sulla gestione delle tabelle e dei dati**
- **Approfondire in autonomia il linguaggio SQL**
- **Riepilogo**

Panoramica

Nel Capitolo 1, avete appreso le basi del linguaggio SQL e avete imparato a interrogare o creare una tabella. In questo capitolo, continuerete a imparare qualcosa di più del linguaggio SQL interrogando e creando più tabelle correlate. Poi vedremo come potete aggiungere altre parole chiave SQL da soli in futuro.

Tabelle correlate

In questo paragrafo imparerete a lavorare con due o più *tabelle correlate*. Si tratta di tabelle di un database che sono collegate fra loro da una o più colonne comuni. The Sci-Fi Collective, per esempio (il negozio online che conoscete dal Capitolo 1), consente ai clienti di crearsi un proprio account, di fare acquisti online e di lasciare recensioni per i prodotti che hanno acquistato. Il negozio di The Sci-Fi Collective è supportato da un database composto da più tabelle. Tra queste ce ne sono due correlate: `product` e `review`. La tabella `product` rappresenta i prodotti venduti e la tabella `review` rappresenta le recensioni che i clienti lasciano per i prodotti che hanno acquistato. Le due tabelle hanno una colonna in comune (`product_id`). La relazione tra le due tabelle è riassunta nella seguente figura:



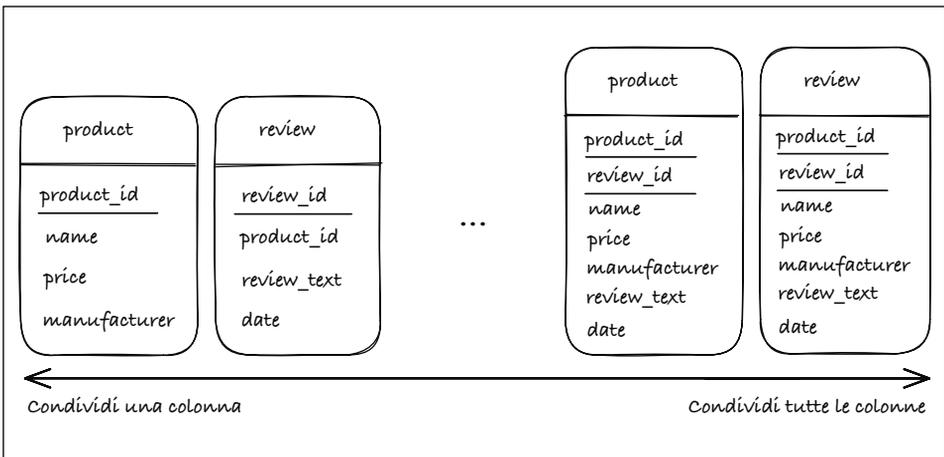
La colonna `product_id` è condivisa dalle tabelle `product` e `review`. Nella tabella `product`, la colonna `product_id` è la chiave primaria. Nella tabella `review`, la colonna `product_id` è nota come *chiave esterna*, in quanto fa riferimento alla chiave primaria di un'altra tabella. In questo caso, l'altra tabella è, appunto, `product`.

In altre parole, il valore nella colonna `product_id` aiuta a collegare una riga della tabella `product` a una riga della tabella `review`. Nella tabella `product`, per esempio, la riga col valore `product_id` pari a 3 è relativa al prodotto *Cat-Poop Coffee*; nella tabella `review`, la riga col valore `review_id` pari a 1 registra una recensione positiva per il prodotto *Cat-Poop Coffee*. Come facciamo a sapere che la recensione è relativa a *Cat-Poop Coffee*? Perché le due righe hanno lo stesso valore `product_id`.

Numero di colonne condivise

Per curiosità, potreste chiedervi se le tabelle `product` e `review` possano condividere anche qualche colonna in più. Questa è un'ottima domanda sulla progettazione del database. Una domanda più generalizzata sarebbe se due tabelle correlate possano condividere anche altre colonne oltre alla chiave primaria/esterna. La risposta è no.

Per semplificare la discussione, esaminiamo lo scenario in cui due tabelle correlate (le tabelle `product` e `review`) utilizzino singole colonne come chiavi primarie. Teoricamente parlando, il numero di colonne condivise da due tabelle correlate può variare dalla sola chiave primaria/esterna a tutte le colonne di entrambe le tabelle. Utilizzando le tabelle `product` e `review` come esempio, la seguente figura riassume questo lungo intervallo:

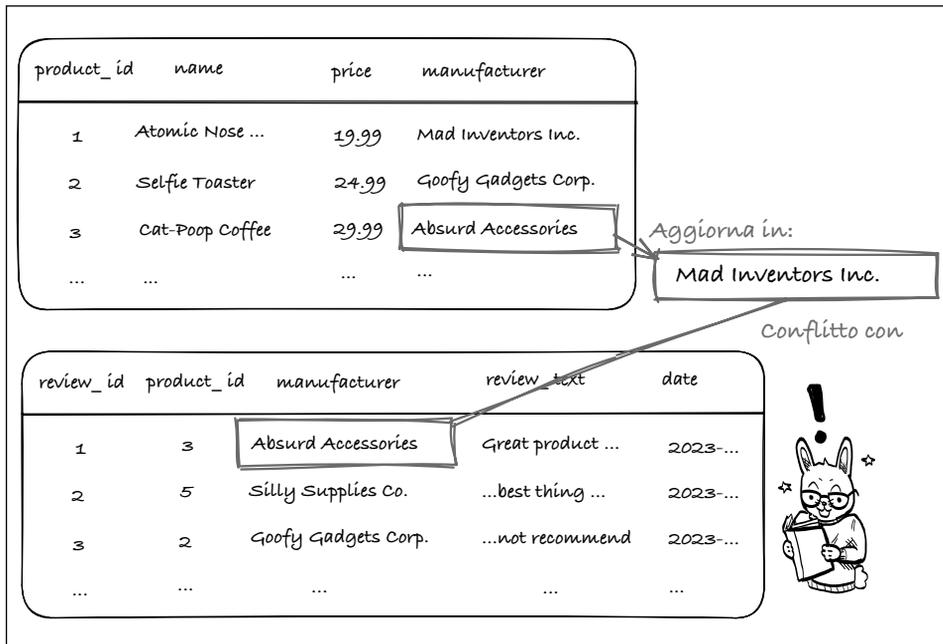


Se due tabelle correlate condividono ogni colonna, è ovvio che una di esse è ridondante e quindi non necessaria. Se scegliete di eliminarne una, vi troverete ad affrontare un problema simile a quello che avete visto nel Capitolo 1: usare una tabella per rappresentare due entità, il che porterà ad anomalie di inserimento o eliminazione. In buona sostanza, far sì che due tabelle condividano ogni colonna è una cattiva idea.

Che ne dite di immaginare che le due tabelle possano condividere solo poche colonne, magari una o due colonne oltre la chiave primaria/esterna? Anche questa è una cattiva idea. Innanzitutto, avreste comunque una ridondanza, anche se meno grave di un'intera

tabella ridondante. In secondo luogo, vi creerete una trappola per quando dovrete aggiornare i dati in tali tabelle. Se aggiornate i dati in una sola tabella e dimenticate l'altra, finirete con avere dati incoerenti.

Supponiamo che decidiate di far sì che le tabelle *product* e *review* condividano un'altra colonna, *manufacturer*, oltre a *product_id*. Quando dovrete aggiornare il produttore di un prodotto, dovrete aggiornare sia la tabella *product* sia la tabella *review*. Altrimenti, finirete per avere i nuovi dati del produttore in una tabella ma i vecchi dati nell'altra. Tale problema è noto come *anomalia di aggiornamento*. Come potete vedere nella seguente figura, due tabelle correlate non devono mai condividere altre colonne oltre la chiave primaria/esterna.



Unione di dati da tabelle correlate

Di tanto in tanto, dovrete mettere insieme dati provenienti da tabelle correlate. Se, per esempio, volete sapere come viene recensito ogni prodotto di The Sci-Fi Collective dovrete unire almeno il nome del prodotto, dalla tabella *product*, e le recensioni corrispondenti dalla tabella *review* e quindi assicurarvi che i dati provenienti da entrambe le tabelle vengano uniti correttamente. In SQL, potete scrivere la query che esegue questo lavoro come segue:

```
SELECT name, review_text
FROM product
JOIN review
ON product.product_id = review.product_id;
```

Questa query produce il seguente risultato.

name	review_text
Cat-Poop Coffee	Great product ...
Unicorn Horn Polish	This is the best ...
Selfie Toaster	Not worth the money ...
...	...

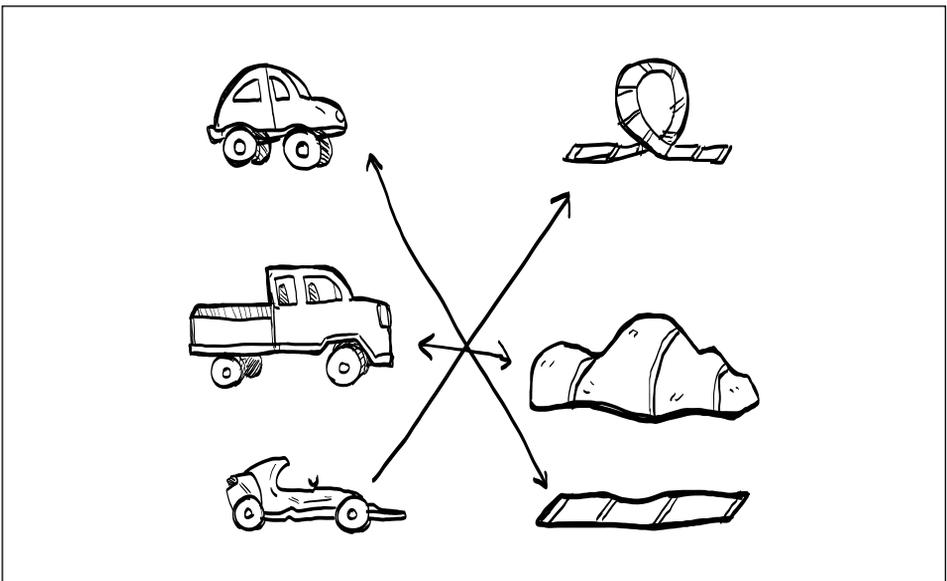
Occorre spiegare alcune cose su questa query.

- Qual è la sintassi generale per recuperare dati da tabelle correlate?
- Che cosa fa la clausola JOIN...ON...?
- Che significato ha il punto e come si usa la notazione col punto in `product.product_id` e `review.product_id`?

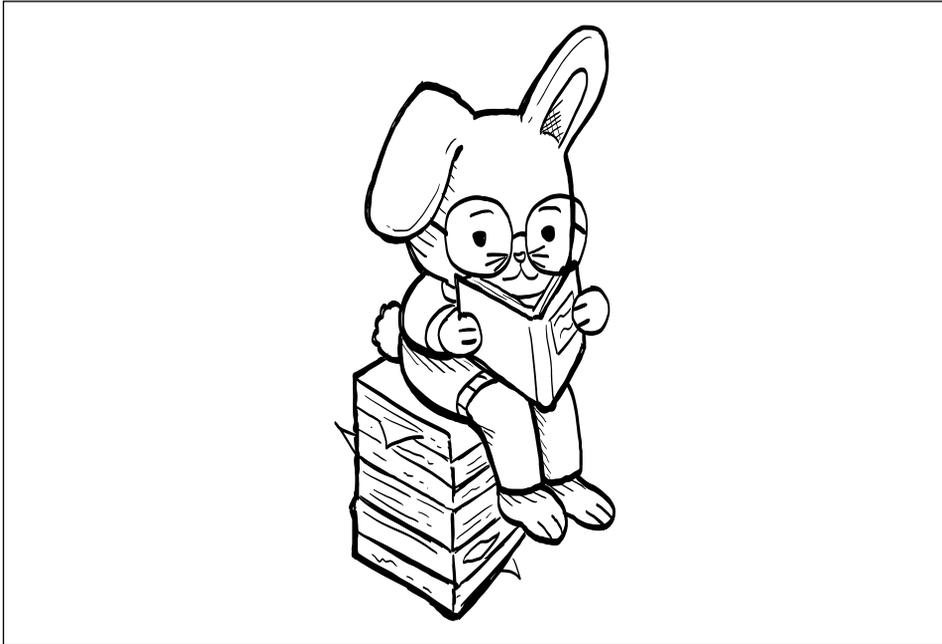
Qual è la sintassi generale per recuperare dati da tabelle correlate? La sintassi generale è la seguente:

```
SELECT colonna1, colonna2, ...
FROM tabella1
JOIN tabella2
ON tabella1.colonna = tabella2.colonna;
```

Che cosa fa la clausola JOIN...ON...? Supponiamo di avere due scatole di giocattoli, una con le macchinine e l'altra con le piste da corsa. Volete giocare con entrambi i giocattoli, quindi rovesciate tutti i giocattoli di entrambe le scatole sul pavimento; poi dovete trovare quali macchinine funzioneranno su quali tipi di piste. Ecco quello che fa la clausola JOIN.



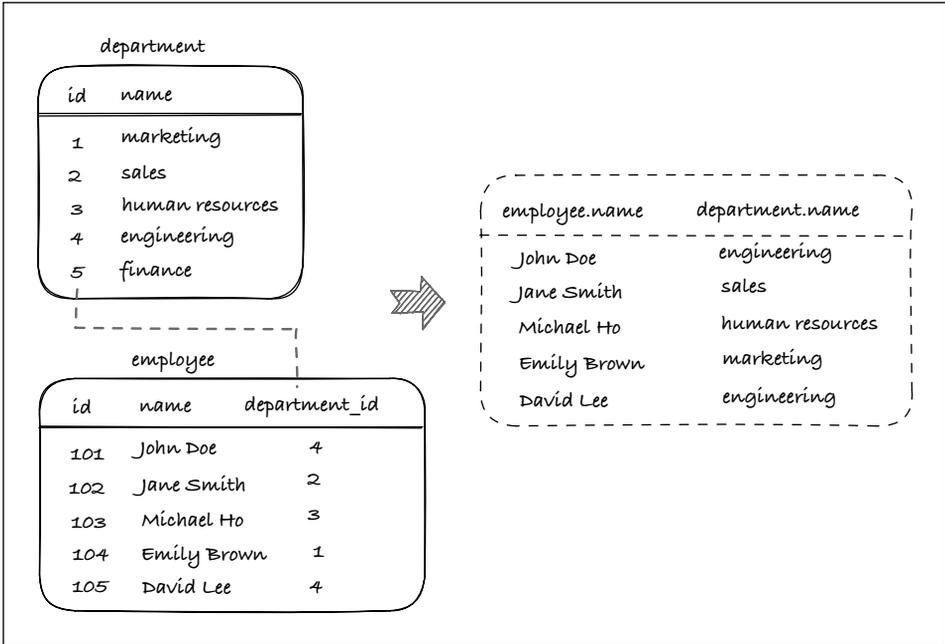
In dettaglio, la clausola `JOIN` prende due tabelle (le scatole dei giocattoli) e ne rovescia tutte le righe (i giocattoli) sul pavimento; poi cerca valori corrispondenti nella colonna condivisa (come la corrispondenza tra un'auto e la sua giusta pista) tra le due tabelle. Se trova una corrispondenza, mette insieme le righe corrispondenti (l'auto e la sua pista) in una nuova tabella (lo spazio di gioco).



Che significato ha il punto e come si usa la notazione a punto come in `product.product_id` e `review.product_id`? La notazione a punto è una sintassi SQL usata per separare le parti di un nome. `product.product_id`, per esempio, fa riferimento alla colonna `product_id` della tabella `product`. Come altro esempio, `product.name` fa riferimento alla colonna `name` della tabella `product`. La notazione a punto è particolarmente utile quando si interrogano tabelle correlate, perché aiuta a essere specifici sulle colonne nel caso in cui abbiano lo stesso nome, come nel caso di `product.product_id` e `review.product_id`. Questo approccio chiarisce a quale colonna e a quale tabella ci si riferisce ed evita ogni confusione.

Se due tabelle che volete unire con `JOIN` hanno le loro colonne con lo stesso nome, potreste volerle rinominare nell'istruzione `SELECT` per evitare confusione. Altrimenti, potreste ritrovarvi con un set di risultati che somiglia a una confusa partita di Scarabeo giocata da un bambino dispettoso.

Supponiamo di avere due tabelle correlate, `employee` e `department`, e di volerle unire con una `JOIN` per ottenere il nome del reparto (`department`) cui appartiene ciascun dipendente (`employee`).



Entrambe le tabelle hanno una colonna denominata *name*, quindi sarà necessario utilizzare la notazione a punto per specificare quale colonna *name* selezionare:

```
SELECT employee.name, department.name
FROM employee
JOIN department
ON employee.department_id = department.id;
```

A seconda del database relazionale che utilizzate, potreste ottenere una colonna, come nella figura precedente, o due colonne dal nome identico (per esempio *name*). Per evitare confusione, potete rinominare le colonne tramite la clausola *AS*:

```
SELECT employees.name AS employee_name, departments.name AS department_name
FROM employees
JOIN departments
ON employees.department_id = departments.id;
```

Tipi di JOIN

Ora che conoscete le basi della clausola *JOIN*, la approfondiremo un po' esaminandone le varianti. Ma prima di farlo, rinfreschiamo la memoria sulla sintassi generale delle operazioni di *JOIN* sulle tabelle:

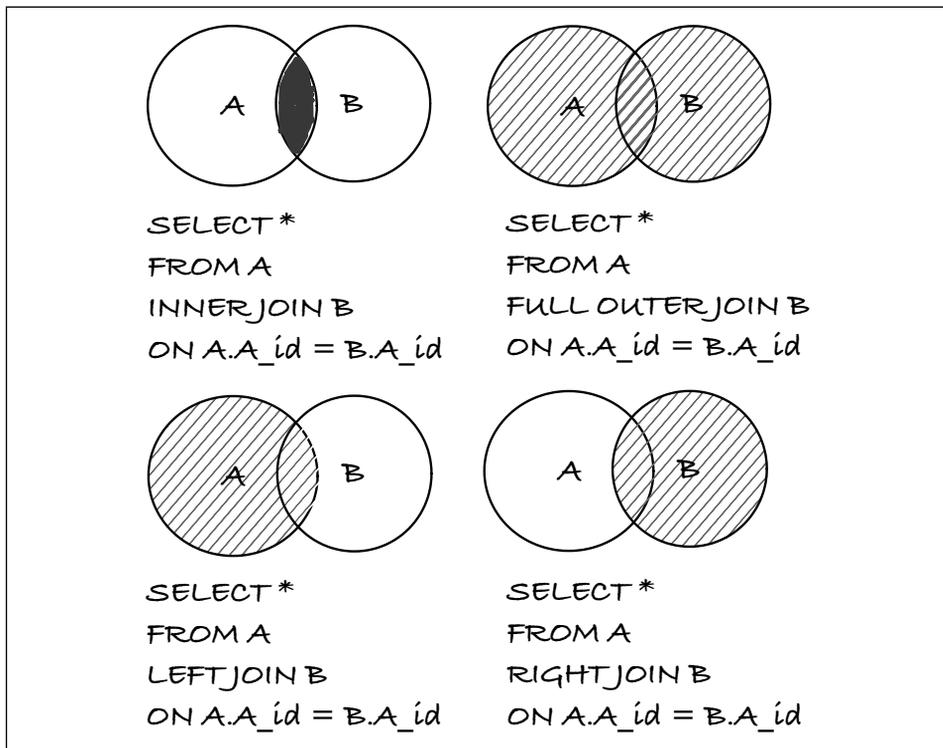
```
SELECT colonna1, colonna2, ...
FROM tabella1
```

```
JOIN tabella2
ON tabella1.colonna = tabella2.colonna;
```

La parola chiave JOIN può essere sostituita dalle seguenti parole chiave, che possono dare origine a risultati differenti.

- INNER JOIN restituisce solo le righe che hanno valori corrispondenti in entrambe le tabelle; sostanzialmente, INNER JOIN è uguale a JOIN.
- LEFT JOIN restituisce tutte le righe della *tabella1* e le righe corrispondenti della *tabella2*. Se un record nella *tabella1* non ha corrispondenze nella *tabella2*, il risultato include il record della *tabella1* con valori NULL per le colonne della *tabella2*.
- RIGHT JOIN restituisce tutte le righe della *tabella2* e le righe corrispondenti della *tabella1*. Se un record della *tabella2* non ha una corrispondenza nella *tabella1*, il risultato include il record della *tabella2* con valori NULL per le colonne della *tabella1*.
- FULL OUTER JOIN restituisce tutte le righe di entrambe le tabelle, incluse le righe non corrispondenti. Se un record nella *tabella1* ha un record corrispondente nella *tabella2*, il risultato include una singola riga con i dati di entrambe le tabelle. Se un record nella *tabella1* non ha corrispondenze nella *tabella2*, il risultato include il record della *tabella1* con valori NULL per le colonne della *tabella2*; se un record nella *tabella2* non ha corrispondenze nella *tabella1*, il risultato include il record della *tabella2* con valori NULL per le colonne della *tabella1*.

Le relazioni tra la tabella di sinistra (left), la tabella di destra (right) e i risultati restituiti sono riassunte graficamente nella seguente figura.



Vale la pena di notare che le query che usano istruzioni `LEFT JOIN`, `RIGHT JOIN` e `FULL OUTER JOIN` possono dare risultati con valori `NULL`. Un effetto collaterale del fatto di ottenere valori `NULL` nei risultati è che è necessario gestirli con attenzione. I valori `NULL` possono causare errori se si tenta di usarli per eseguire calcoli o confronti. Per esempio, `10 + NULL` sarà ancora uguale a `NULL`. La nostra tabella `product`, per esempio, contiene alcuni nuovi prodotti che non sono ancora stati recensiti da nessuno. Se eseguiamo una `LEFT JOIN` tra le tabelle `product` e `review`, finiamo per avere alcune righe che hanno valori `NULL` nelle colonne della tabella `review`. La query `LEFT JOIN` sarebbe:

```
SELECT name, review_text
FROM product
LEFT JOIN review
ON product.product_id = review.product_id;
```

Questa query produce il seguente risultato.

name	review_text
<i>Atomic Nose Hair Trimmer</i>	Decent product for ...
<i>Atomic Nose Hair Trimmer</i>	Would not recommend ...
...	...
<i>The Mind Probe</i>	NULL
<i>The Sonic Screwdriver</i>	NULL
...	...

Come potete vedere, troviamo ogni corrispondenza tra i prodotti e le recensioni. Un prodotto come *Atomic Nose Hair Trimmer* può avere più recensioni e comparire più di una volta nella tabella dei risultati, sotto forma di più righe. Ma anche se un prodotto non è mai stato recensito, viene comunque incluso nella tabella dei risultati. È il caso di *The Mind Probe*, che in `review_text` ha valore `NULL`.

WHERE vs. JOIN

Per curiosità, potreste essere tentati di provare a unire (con `JOIN`) due tabelle usando la clausola `WHERE`. Nel Capitolo 1 avete imparato a usare la clausola `WHERE` per estrarre da una tabella un sottoinsieme di dati che soddisfa determinati criteri. Se sapete che potete elencare più tabelle nell'istruzione `FROM`, potreste mettere insieme la seguente query per unire le tabelle `product` e `review` che abbiamo appena visto.

```
SELECT name, review_text
FROM product, review
WHERE product.product_id = review.product_id;
```

Questa query funzionerebbe e produrrebbe lo stesso risultato dell'esempio che abbiamo visto prima? Sì. Questa query funzionerà bene per unire le due tabelle e produrrà lo stesso risultato della query che usa la clausola `JOIN`.

```
SELECT name, review_text
FROM product
JOIN review
ON product.product_id = review.product_id;
```

Tuttavia, ogni volta che è necessario interrogare più tabelle correlate, si preferisce usare `JOIN` rispetto a `WHERE` per almeno tre motivi.

- *Leggibilità*: una `JOIN` esplicita rende più chiaro e facile da comprendere l'intento della query.
- *Manutenibilità*: una `JOIN` esplicita è meno soggetta a errori e più semplice da modificare o sottoporre a debug.
- *Ottimizzazione*: quando si utilizza una clausola `WHERE` per interrogare due tabelle correlate, la query richiede essenzialmente una `JOIN` incrociata (*cross join*) tra le due tabelle, che per la maggior parte dei database relazionali risulta più difficile da ottimizzare rispetto a una `JOIN` esplicita.

Cross join: una folle danza in cui tutti volteggiano con tutti gli altri

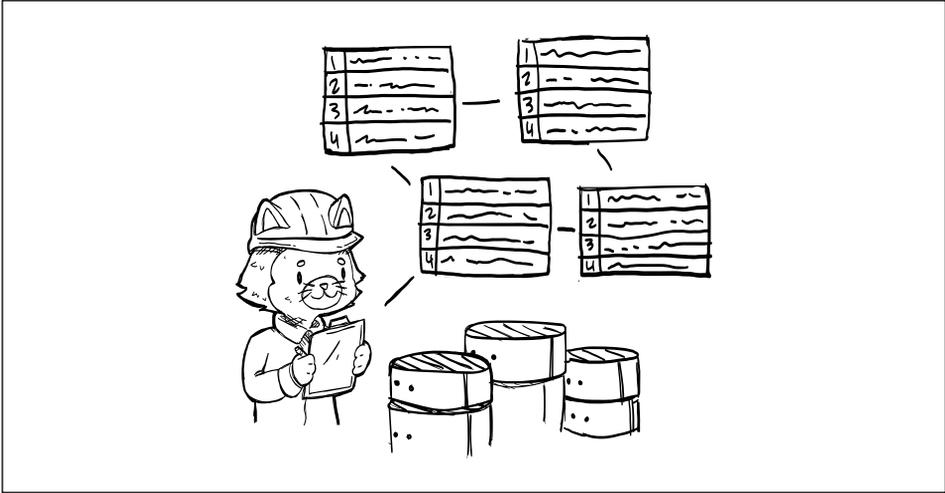
Una *cross join* SQL è un'operazione che combina ogni singola riga di una tabella con ogni singola riga di un'altra tabella e genera tutte le possibili combinazioni di righe tra le due tabelle, restituendo un risultato enorme.

Se la clausola `FROM` della vostra query è seguita da due o più tabelle, la vostra query eseguirà una *join* incrociata tra quelle tabelle. `FROM product, review`, per esempio, fa in modo che ogni singola riga della tabella `product` venga abbinata a ogni singola riga della tabella `review`, indipendentemente dal fatto che esista o meno una corrispondenza.

Le *cross join* potrebbero richiedere una scansione separata delle tabelle interessate e impegnano una grande quantità di memoria sulla macchina in cui risiede il database relazionale.

Di nuovo sulla gestione delle tabelle e dei dati

Questo paragrafo spiega come gestire le tabelle correlate e i loro dati. Nel Capitolo 1, avete imparato a gestire una singola tabella e i suoi dati. Nel paragrafo precedente, avete lavorato con una coppia di tabelle correlate. Ora applicherete ciò che avete imparato per ampliare la vostra conoscenza alla gestione delle tabelle correlate e dei loro dati.



Gestione di tabelle correlate

Per imparare a creare tabelle correlate impiegheremo gli script che accompagnano questo capitolo. Tali script creano due tabelle correlate, *product* e *review*, per il database che supporta lo shop online The Sci-Fi Collective e a popolarle con un set di dati campione. Scegliete lo script che funziona col vostro database relazionale, ovviamente (per esempio, *mysql_db.sql* per MySQL o MariaDB).

La tabella *product* viene creata nello stesso modo già visto nel Capitolo 1.

```
CREATE TABLE product (
  product_id INT PRIMARY KEY,
  name TEXT NOT NULL,
  description TEXT NOT NULL,
  price DECIMAL(5, 2) NOT NULL,
  manufacturer TEXT NOT NULL
);
```

Per creare la tabella *review*, procedete come segue:

```
-- Commento - Funziona con MySQL e MariaDB
-- Commento - Nel repository del codice trovate altre versioni per altri RDBMS
CREATE TABLE (
  review_id INT PRIMARY KEY,
  product_id INT NOT NULL,
  review_text TEXT NOT NULL,
  datetime DATETIME NOT NULL
  DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT fk_product_review
  FOREIGN KEY (product_id)
  REFERENCES product (product_id)
);
```

Dobbiamo rispondere a due domande relative a questa query.

- Qual è la sintassi generale per creare due tabelle che sono in relazione fra loro?
- Che cosa fa la clausola `CONSTRAINT ... FOREIGN KEY ... REFERENCES ...`?

Qual è la sintassi generale per creare due tabelle che sono in relazione fra loro? Come sapete, la colonna condivisa `product_id` è la chiave primaria nella tabella `product` e la chiave esterna della tabella `review`. Data una coppia di tabelle correlate, chiamiamo una tabella come `product`, usa la colonna condivisa come chiave primaria, *tabella genitore*. Analogamente, chiamiamo una tabella come `review`, che contiene la chiave esterna, *tabella figlia*.



Come potete vedere dal comando che crea la tabella `product`, la sintassi per creare una tabella genitore è la stessa usata per creare una tabella non correlata ad altre tabelle. Per creare una tabella figlia, invece, dovete specificare la chiave esterna che fa riferimento alla chiave primaria della tabella genitore. Dovete comunque definire anche tutte le colonne, i tipi di dati e la chiave primaria, ovviamente. La sintassi generale per creare una tabella figlia è la seguente:

```
CREATE TABLE nome_tabella_figlia (
  colonna1 tipo_di_dati1 [NOT NULL],
  colonna2 tipo_di_dati2 [NOT NULL],
  ...,
  colonna_della_chiave_esterna tipo_di_dati,
  CONSTRAINT chiave_esterna_genitore_figlia
    FOREIGN KEY (colonna_della_chiave_esterna)
    REFERENCES tabella_genitore (chiave_primaria_della_tabella_genitore)
);
```

In alternativa, potete creare le due tabelle in modo indipendente e aggiungere solo successivamente il vincolo di chiave esterna alla tabella figlia:

```
-- Commento - Presupponendo che le tabelle genitore e figlia siano già state create
ALTER TABLE nome_tabella_figlia
  ADD CONSTRAINT chiave_esterna_genitore_figlia
    FOREIGN KEY (colonna_della_chiave_esterna)
    tabella_genitore (chiave_primaria_della_tabella_genitore);
```

Che cosa fa la clausola `CONSTRAINT ... FOREIGN KEY ... REFERENCES ...`? In breve, la clausola crea un vincolo di chiave esterna, che funge da collegamento tra due tabelle correlate. Il vincolo è duplice:

- garantisce che la colonna della chiave esterna della tabella figlia faccia riferimento solo a valori della chiave primaria che sono validi nella tabella genitore;
- garantisce che l'aggiornamento o l'eliminazione di righe dalla tabella genitore non violi la coerenza tra le due tabelle correlate.



Questi due aspetti vanno sotto il nome cumulativo di *integrità referenziale*. La clausola `CONSTRAINT ... FOREIGN KEY ... REFERENCES ...` impone l'integrità referenziale tra due tabelle correlate. Osservando più da vicino la clausola `CONSTRAINT ... FOREIGN KEY ... REFERENCES ...`, potete dividerla in due parti, come mostrato nella seguente figura.



L'istruzione `FOREIGN KEY ... REFERENCES ...` crea il vincolo di chiave esterna che impone l'integrità referenziale. La clausola `CONSTRAINT ...` consente di dare un nome a questo vincolo. Quando si crea un vincolo di chiave esterna, non è necessario dargli un nome, ma il fatto che ha un nome renderà più facile l'accesso ogni volta che in futuro occorrerà modificare tale vincolo. Per esempio, nella tabella `review` abbiamo chiamato il vincolo

alla chiave esterna `fk_product_review`. Se in futuro dovessimo eliminare questo vincolo, potremo accedervi tramite tale nome:

```
-- Commento - SQLite non supporta ALTER TABLE DROP CONSTRAINT
ALTER TABLE review
  DROP CONSTRAINT fk_product_review;
```

Se non date un nome a un vincolo, il database relazionale gli assegnerà automaticamente un nome usando una sua convenzione standard. Sebbene tale nome, scelto automaticamente, possa essere recuperato, tale nome e la convenzione di denominazione variano da un database relazionale all'altro. Per evitare questo fastidio, vi consigliamo di dare sempre un nome ai vincoli (come si è soliti dare un nome agli animali domestici). Dopotutto, eliminare o disabilitare un vincolo può essere un'attività comune, quando dovrete modificare la struttura del database.

Gestione dei dati nelle tabelle correlate

La sintassi per aggiungere, aggiornare ed eliminare i dati nelle tabelle correlate rimane la stessa che avete visto nel Capitolo 1. Gli script che abbiamo preparato per questo capitolo aggiungono alcuni dati sia alla tabella `product` sia alla tabella `review`:

```
-- Commento - Aggiunge dati alla tabella product
INSERT INTO product (product_id, name, description, price, manufacturer)
VALUES (
  1,
  'Atomic Nose Hair Trimmer',
  'Trim your nose hairs ... an atomic clock!',
  19.99,
  'Mad Inventors Inc.'
),
...;

-- Commento - Aggiunge dati alla tabella review
INSERT INTO review (review_id, product_id, review_text, datetime)
VALUES (
  1,
  3,
  'Great product, would definitely recommend!',
  '2022-01-01 12:30:00'
),
...;
```

Ciò che rende diversa la gestione dei dati nel caso di tabelle correlate è il vincolo alla chiave esterna. In precedenza, in questo capitolo, avete imparato che il vincolo alla chiave esterna impone l'integrità referenziale su una coppia di tabelle correlate:

- il vincolo di chiave esterna garantisce che la colonna della chiave esterna della tabella figlia faccia riferimento solo a valori validi della chiave primaria della tabella genitore;
- il vincolo di chiave esterna garantisce che l'eliminazione di righe dalla tabella genitore non violi la coerenza tra le due tabelle correlate.

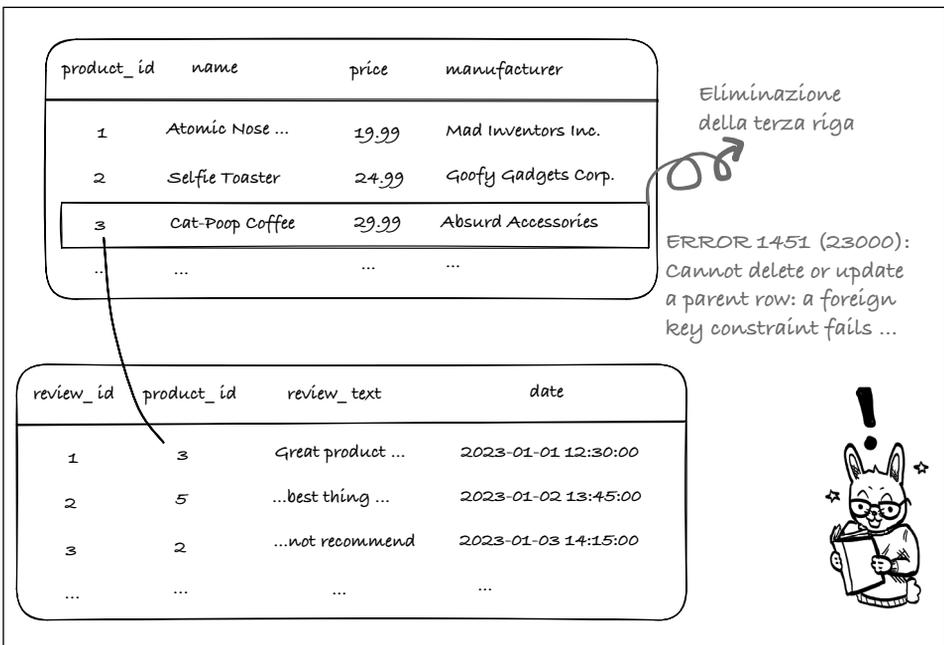
Quando si tenta di aggiungere nuovi dati alla tabella figlia, tali dati devono essere coerenti con quelli già esistenti nella tabella genitore; in caso contrario, il database relazionale se ne lamenterà. Supponiamo che stiate tentando di aggiungere una nuova recensione (una riga di dati) alla tabella `review`, ma che il valore `product_id` (per esempio 3000) di questa riga non esista nella tabella `product`:

```
INSERT INTO revisione (review_id, product_id, review_text, datetime)
VALUES (
  1,
  3000,
  'Great product!',
  '2023-05-01 12:30:00'
);
```

Quando eseguirete questo comando, il vostro database relazionale vi risponderà con un messaggio d'errore simile al seguente:

```
ERROR 1452 (23000):
Cannot add or update a child row: a foreign key constraint fails ...
```

Allo stesso modo, quando modificate o eliminate dei dati dalla tabella genitore, tale operazione non dovrebbe creare record di dati orfani nella tabella figlia; se ciò accade, il vostro database relazionale se ne lamenterà. Supponiamo che vogliate eliminare una riga di dati (un prodotto) dalla tabella `product`, ma che questo prodotto abbia una recensione nella tabella `review`. Se eliminate questa riga, l'eliminazione creerà alcune recensioni orfane nella tabella `review`. Fortunatamente, il vincolo a chiave esterna impedirà che questa situazione si verifichi, e riceverete un messaggio d'errore simile al seguente.



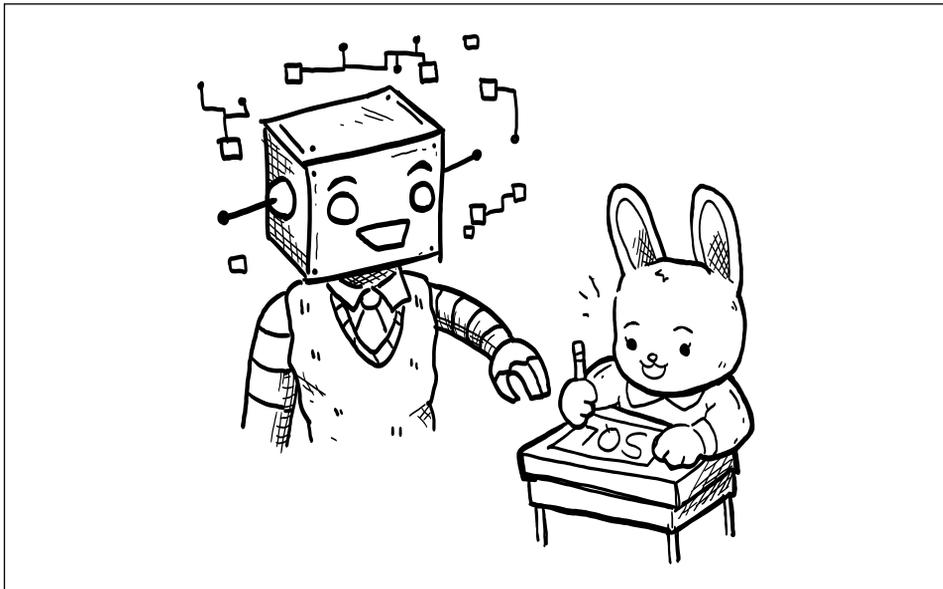
La forma più drastica di eliminazione di dati dalla tabella genitore consiste nell'eliminazione completa della tabella. Se questa azione fosse consentita, diverrebbero orfani tutti i dati presenti nella tabella figlia. Fortunatamente, il vincolo alla chiave esterna impedisce che ciò accada e otterrete un messaggio d'errore simile al seguente:

```
ERROR: cannot drop table ... because other objects depend on it
DETAIL: constraint... on table... depends on table...
```

Se state ancora perfezionando la struttura stessa del database e avete bisogno di modificare la struttura della tabella, potete fare riferimento al Capitolo 1 per vedere come modificare una singola tabella, oppure vedere il paragrafo precedente per scoprire come modificare il vincolo alla chiave esterna. In uno scenario estremo, potreste dover eliminare una coppia di tabelle correlate con alcuni dati di esempio. In questo caso, salvate sempre prima i dati ed eliminate prima la tabella figlia e poi la tabella genitore. Questo ordine è importante, perché eliminare prima la tabella figlia elimina anche il vincolo alla chiave esterna, e otterrete il messaggio d'errore della figura precedente, perché il vincolo alla chiave esterna ha proprio lo scopo di far rispettare l'integrità referenziale.

Approfondire in autonomia il linguaggio SQL

In questo paragrafo, vi offriremo alcuni suggerimenti per proseguire lo studio del linguaggio SQL da soli. Il viaggio verso la padronanza del linguaggio SQL è lungo, ma in tal modo avrete un sacco di tempo per riflettere su tutti i modi in cui avete scritto male un `SELECT` o avete sbagliato con un `NULL`. Tuttavia, se avete letto il Capitolo 1 e questo capitolo, avete già gli elementi essenziali per progettare un database. Queste conoscenze saranno per voi una solida base per approfondire la vostra conoscenza del linguaggio SQL.



Al giorno d'oggi, imparare da soli il linguaggio SQL è molto più facile di quanto non fosse qualche anno fa, grazie all'ascesa degli strumenti di intelligenza artificiale generativa come ChatGPT e Google Gemini. Non fraintendeteci: non stiamo sostenendo di affidarsi esclusivamente a ChatGPT per imparare a usare il linguaggio SQL. Se ne usati, gli strumenti di intelligenza artificiale generativa come ChatGPT permettono di acquisire nuove conoscenze in modo rapido ed efficiente, e vogliamo dimostrarvi come potete usarli per facilitare l'apprendimento del linguaggio SQL.

Alcuni aspetti dell'apprendimento restano ovviamente gli stessi del passato. Avrete bisogno dell'aiuto di un buon libro per imparare a usare il linguaggio SQL in modo sistematico, in particolare per quanto concerne conoscenze concettuali come l'indicizzazione e l'ordine di esecuzione delle query. Questo passaggio non può essere sostituito ponendo domande a ChatGPT. Sono disponibili molti buoni libri sull'argomento, fra cui:

- *SQL*, Marco Ferrero, <https://www.apogeeonline.com/libri/sql-marco-ferrero/>;
- *Basi di dati*, Serena Sensini, <https://www.apogeeonline.com/libri/basi-di-dati-serena-sensini/>.

Tuttavia, altri aspetti dell'apprendimento del linguaggio SQL possono essere molto diversi e più efficienti, oggi, specialmente se avete domande o volete sapere come viene utilizzata una determinata parola chiave. Il più grande vantaggio degli strumenti di intelligenza artificiale generativa come ChatGPT è che possono fornire un aiuto relativamente affidabile e tempestivo sulle domande più comuni. Senza tali strumenti, dovete seguire un processo iterativo per ottenere le risposte alle vostre domande, il che implica la formulazione di un testo per la ricerca, poi la ricerca online e infine il filtraggio dei risultati restituiti dalla ricerca. Questo processo, specialmente il filtraggio di tutti i risultati, può essere scoraggiante.

L'ampia disponibilità di strumenti di intelligenza artificiale generativa può aiutarvi a evitare questo processo iterativo. Se, per esempio, volete sapere come usare la funzione di somma aggregata, potete porre a ChatGPT la seguente domanda:

Come si usa la funzione di somma aggregata in SQL?

La risposta, solitamente, è composta dalla sintassi generale e da un esempio concreto, che è valido quanto, e a volte più, di alcuni tutorial scritti da esseri umani.

Ecco di seguito un esempio di risposta fornita da ChatGPT.

ChatGPT

In SQL, la funzione SUM viene utilizzata per calcolare il totale di un set di valori in una colonna di una tabella. Ecco la sintassi di base:

```
SELECT SUM(nome_colonna)
FROM nome_tabella;
```

Per esempio, supponiamo di avere una tabella chiamata `Ordini` con le seguenti colonne: `Id`, `Nome_cliente`, `Nome_prodotto` e `Prezzo`. Per calcolare il fatturato totale di tutti gli ordini, potresti usare la seguente query:

```
Select Sum(Prezzo)
From Ordini;
```

Ciò restituirebbe un singolo valore, che rappresenta la somma di tutti i prezzi riportati nella colonna Prezzo della tabella Ordini.

Se la risposta vi sembra ancora troppo ostica, potete chiedere a ChatGPT di essere paziente e di spiegarvela più in dettaglio:

Come si usa la funzione somma aggregata in SQL? Spiegalo con più esempi e come se io avessi cinque anni.

Scoprirete che ChatGPT risponde a questa domanda meglio di quanto possa fare la maggior parte dei tutor. Ecco un esempio di risposta da ChatGPT.

ChatGPT

Certo! Diciamo che hai una tabella contenente alcuni numeri, come questa:

```
+----+-----+
| ID | Valore |
+----+-----+
|  1 |      2 |
|  2 |      4 |
|  3 |      6 |
|  4 |      8 |
+----+-----+
```

Ogni riga ha un numero nella colonna Valore. Immaginate che questi numeri siano caramelle, di vari gusti. La colonna ID è come un'etichetta che ci aiuta a distinguere i vari gusti delle caramelle. Ora, diciamo che vuoi sapere quante caramelle hai in totale. Potresti contare ogni caramella una per una, ma impieghereste molto tempo. Invece, puoi usare la funzione SUM di SQL perché sommi tutte le caramelle per te! Ecco come sarebbe la query SQL:

```
SELECT SUM(Valore) FROM tabella;
```

Quando esegui questa query, il computer somma tutti i numeri che ci sono nella colonna Valore. In questo caso, il totale è 20. Hai scoperto di avere 20 caramelle in totale! Ecco un altro esempio. Diciamo che hai una tabella con alcuni prezzi, come questa:

```
+----+-----+
| ID | Prezzo |
+----+-----+
|  1 |   1.99 |
|  2 |   2.99 |
|  3 |   3.99 |
|  4 |   4.99 |
+----+-----+
```

Ogni riga ha un prezzo nella colonna Prezzo. Puoi pensare che questi siano i prezzi di giocattoli o caramelle. Con la funzione SUM di SQL puoi sapere quanti soldi ti servirebbero per acquistare tutte queste cose. Ecco come sarebbe la query:

```
SELECT SUM(Prezzo) FROM tabella;
```

Quando esegui questa query, il computer somma tutti i prezzi contenuti nella colonna Prezzo. In questo caso, il totale è 13,96. Questa è la somma che devi avere in tasca per comprare tutte queste cose!

Uno dei vantaggi più significativi degli strumenti di intelligenza artificiale generativa nell'apprendimento di un linguaggio come SQL è che possono fornirvi un aiuto “al volo” e relativamente affidabile. Per le domande più comuni, non è necessario porre una domanda perfetta per ricevere una risposta veloce, informativa e ben strutturata. Inoltre, a differenza di quanto accade per la composizione delle domande per i motori di ricerca, è più facile mettere a punto le domande per ottenere risposte migliori.



Riepilogo

- Le tabelle correlate di un database sono collegate in base a una colonna comune.
- In generale, non è bene che tabelle correlate condividano altre colonne, oltre alla chiave primaria/esterna.
- La sintassi generale per interrogare due tabelle correlate è la seguente:

```
SELECT colonna1, colonna2, ...
FROM tabella1
```