Home | MySQL | MySQL Tutorial 4: Relational Databases Share This Page

MySQL Tutorial 4: Relational Databases

How to build a modern database

— P. Lutus — Message Page —

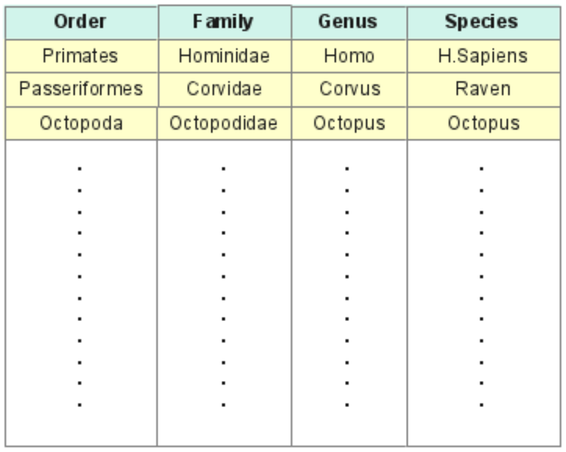
(double-click any word to see its definition)

— Navigate this multi-page article with the arrows and drop-down lists at the top and bottom of each page —

Note: In this article, footnotes are marked with a light bulb over which one hovers .

Introduction

A relational database differs from a conventional database by having connections between tables that define relationships. Before relational database ideas are applied, absent any defined relationships, a database table might look like this:



Order	Family	Genus	Species
Primates	Hominidae	Homo	H.Sapiens
Passeriformes	Corvidae	Corvus	Raven
Octopoda	Octopodidae	Octopus	Octopus
⋮	⋮	⋮	⋮

Figure 1: Flat table database

The drawback to such a system is that, as records are added, errors can and will be made — a species may not belong to the genus to its left, or a genus to the family to its left, *ad infinitum*, and unless someone carefully researches each entry, errors are inevitable.

A relational database solves these problems by strictly ordering the relationship between the levels, thus preventing the entry of a species not appropriate to a given genus, or genus to a family, and so forth. Until recently such an arrangement was dearly wished for but not practical. But with a hierarchical database design based on relational principles, a hierarchy with a degree greater than my simplified diagram is possible, that can strictly relate Domain → Kingdom → Phylum → Class → Order → Family → Genus → Species.

To be more specific, the user of such a database would begin at the leftmost category and choose the desired entry, then move to the right, gradually focusing in on a particular species, allowing the relational database to assist in the process and eliminate errors:

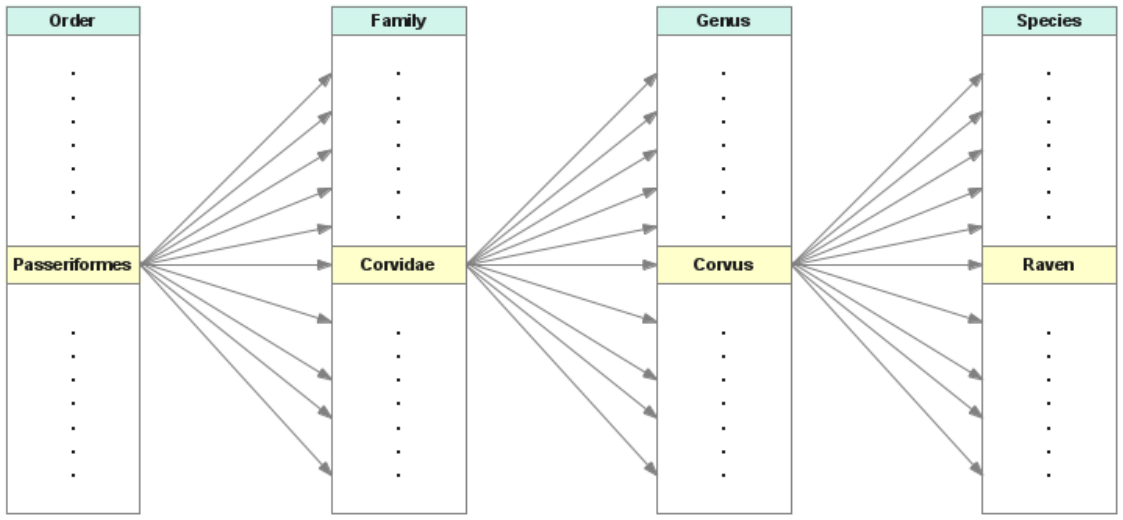


Figure 2: Hierarchical relational database

Also, although less efficiently, using such a system a person could enter a species — the rightmost tier of the taxonomic hierarchy — and let the database engine complete the identification automatically.

The above graphic shows the basic idea of a relational database — it's a strict relationship between tables of data able to produces results that other kinds of data orderings cannot.

In a relational database, redundancies are eliminated — a particular item of information is located at precisely one location, and is linked (joined) to all uses of that item. But the links are references, not copies, which means if the original item needs updating or is found to be in error, only one change needs to be made, and all the uses for that datum change automatically.

For example, in 1996 the Baltimore Oriole was recognized as a separate species . In a relational database system, this would be managed by adding a single entry to the table for the genus *Icterus*. From a practical standpoint, what this means is that the task of classification now has the absolute minimum of complexity — one species: one table entry — and all uses of that information automatically follow.

Relational Inventory Control System

This may sound counterintuitive at first, but the point of relational databases is not to make things more complicated, but simpler and easier to verify and maintain.

A common relational design is that for an inventory control system — such a system has a strict relationship between a table describing an inventory, and tables that define adding to, and removing from, the inventory. For this tutorial I've designed a very simple relational database (relational_demo.sql) with just three tables — an inventory, a sales staff,

and a sales invoice that relies on the others. When I examine my sample database in MySQL Workbench , it looks like this:

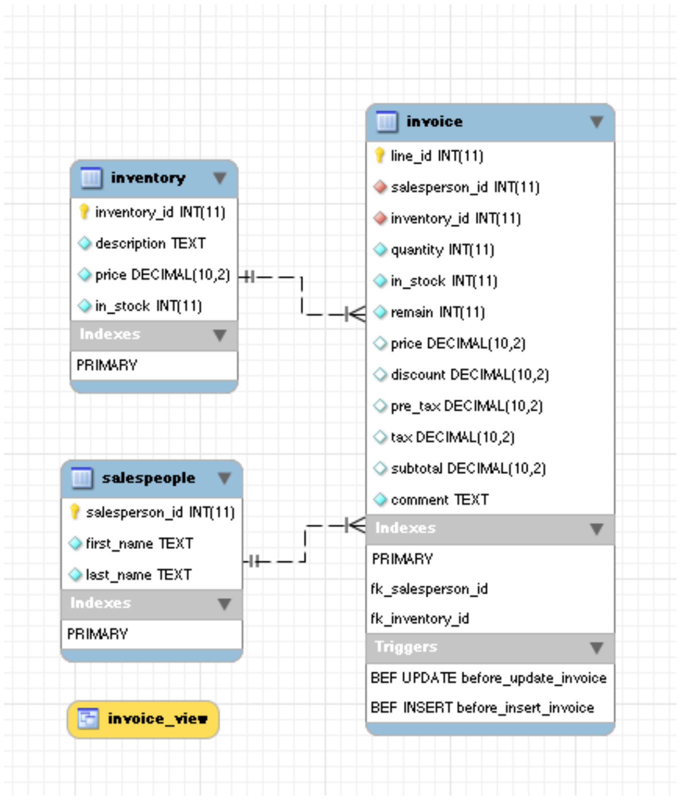


Figure 3: MySQL Workbench model diagram

In Figure 3, the dotted lines connecting the salespeople and inventory tables to the invoice table describe a one-to-many relationship, on the assumption that salespeople and inventory items will appear in more than one invoice record.

This example database is much like that shown earlier, except that it exercises strict control over inventory and salespeople — both can appear in an invoice, but neither of them can be changed by an invoice creator, with a single exception — the stock item quantities that appear on an invoice are automatically subtracted from the inventory. Here are the three tables as they appear after the transaction defined in the demonstration file relational_demo.sql:

salesperson_id	first_name	last_name
1	John	Wilson
2	Terry	Philips
3	Mark	Johnson
4	Mary	Jones
5	Frank	Smith

Figure 4: Salespeople table

inventory_id	description	price	in_stock
1	Mousetrap	5.98	10
2	Hammer	12.33	10
3	Bird Seed	6.44	8
4	Hacksaw	18.23	10
5	Garden Hose	17.43	2
6	Trash Can	14.45	2
7	Nail 6d	0.07	188
8	Nail 12d	0.28	76

Figure 5: Inventory table post-transaction

line_id	salesperson_id	inventory_id	quantity	in_stock	remain	price	discount	pre_tax	tax	subtotal	comment
1	1	1	10	20	10	5.98	0.00	59.80	5.08	64.88	
2	2	2	20	10	10	12.33	0.00	0.00	0.00	0.00	Insufficient stock
3	3	3	30	8	8	6.44	0.00	0.00	0.00	0.00	Insufficient stock
4	4	4	10	20	10	18.23	0.00	182.30	15.50	197.80	
5	5	5	8	10	2	17.43	0.00	139.44	11.85	151.29	
6	4	6	14	16	2	14.45	10.00	182.07	15.48	197.55	
7	3	7	12	200	188	0.07	10.00	0.76	0.06	0.82	
8	2	8	24	100	76	0.28	10.00	6.05	0.51	6.56	

Figure 6: Invoice table post-transaction

Salesperson	Item	Quantity	In Stock	Remain	Price	Discount %	Pretax	Tax	Subtotal	Comment
John Wilson	Mousetrap	10	20	10	5.98	0.00	59.80	5.08	64.88	
Terry Philips	Hammer	20	10	10	12.33	0.00	0.00	0.00	0.00	Insufficient stock
Mark Johnson	Bird Seed	30	8	8	6.44	0.00	0.00	0.00	0.00	Insufficient stock

Mary Jones	Hacksaw	10	20	10	18.23	0.00	182.30	15.50	197.80	
Frank Smith	Garden Hose	8	10	2	17.43	0.00	139.44	11.85	151.29	
Mary Jones	Trash Can	14	16	2	14.45	10.00	182.07	15.48	197.55	
Mark Johnson	Nail 6d	12	200	188	0.07	10.00	0.76	0.06	0.82	
Terry Philips	Nail 12d	24	100	76	0.28	10.00	6.05	0.51	6.56	

Figure 7: Invoice view post-transaction

Look carefully at figures 4,5 and 6 (created using JDBClient) — notice that the salespeople table has a key named "salesperson_id", the inventory table has a key named "inventory_id", and the invoice table has two keys with the same names. This is not a coincidence — the invoice table's use of these key names (plus some additional statements discussed below) cause a strict relationship to exist between the three tables.

The invoice table is linked with the salespeople and inventory tables by what are called "foreign keys" , and these keys control how the invoice functions. It means the invoice can include only salespeople that appear in the salespeople table, and can only include inventory items that appear in the inventory table. And perhaps more important, when an item and a quantity are added to the invoice, a stock check is carried out to make sure there is sufficient stock to fill the order — and if not, the transaction fails with a message.

Two constraint statements control how the two invoice table foreign keys function:

```
constraint `fk_salesperson_id` foreign key (salesperson_id)
    references salespeople (salesperson_id)
    on delete restrict on update restrict,
  constraint `fk_inventory_id` foreign key (inventory_id)
    references inventory (inventory_id)
    on delete restrict on update restrict
```

These specific constraints prevent the user of the inventory table from either deleting from, or changing, the inventory or salespeople tables.

As with the earlier invoice program, this invoice table is equipped with triggers to automatically complete transactions when a record is inserted or edited. Here is the function the triggers call:

```
create procedure proc_process_invoice (
   in salesperson_id integer,
   in inv_id integer,
   in quantity integer,
   inout in_stock integer,
   inout remain integer,
   inout price decimal(10,2),
   inout discount decimal(10,2),
   inout pre_tax decimal(10,2),
   inout tax decimal(10,2),
   inout subtotal decimal(10,2),
   inout comment text
 )
 begin
  select x.in_stock,x.price into in_stock,price from inventory as x
    where inventory_id = inv_id;
  if(in_stock - quantity >= 0) then -- okay to complete transaction
    set discount = if(quantity >= 12,10,0);
    set pre_tax = (price * quantity) * (1- (discount / 100));
    set tax = pre_tax * 0.085;
    set subtotal = pre_tax + tax;
    set remain = in_stock - quantity;
    update inventory set in_stock = remain where inventory_id = inv_id;
  else -- not enough of this item in stock
    set remain = in_stock;
    set comment = 'Insufficient stock';
  end if;
 end|
```

The code above, executed at each invoice table insertion or update, checks stock against the desired purchase quantity and decides whether the transaction can proceed. If it can, various computations are carried out -- a purchase price is established based on the item price times the quantity, a 10% discount is applied if the quantity ordered is 12 or greater, and a sales tax is computed. Finally, the ordered quantity is subtracted from the inventory.

In a practical embodiment of the above code, meant for actual service, there would be a few additional features — deleted inventory items would restore the stock quantity to its previous state, and ongoing transactions would be accompanied by a lock to prevent a simultaneous-transaction race condition (one in which two independent transactions see adequate stock levels, but the sum of the transactions is greater than the inventory can support).

Here is one of the triggers that call the above code (the triggers are virtually identical):

```
CREATE TRIGGER `before_update_invoice`
before update ON `invoice`
for each row
BEGIN
  call proc_process_invoice(
    new.salesperson_id,
    new.inventory_id,
    new.quantity,
    new.in_stock,
    new.remain,
    new.price,
    new.discount,
    new.pre_tax,
    new.tax,
    new.subtotal,
    new.comment);
END|
```

The point of calling a procedure from the trigger is that both triggers (for insert and update) need to do the same thing in the same way, so they call a common code block.

In relational databases more than anywhere else, the value of views become clear — the invoice view (Figure 7) can access a table like the invoice, linked by foreign keys to the other two tables but bereft of most information in text form, and acquire the text representations of various quantities that only appear as indices in the invoice table. Here's the code behind the view:

```
create or replace view invoice_view as select
  concat(se.first_name,'',se.last_name) as Salesperson,
  inv.description as Item,
  iv.quantity as Quantity,
  iv.in_stock as `In Stock`,
  iv.remain as `Remain`,
  iv.price as Price,
  iv.discount as `Discount %`,
  iv.pre_tax as Pretax,
  iv.tax as Tax,
  iv.subtotal as Subtotal,
  iv.comment as Comment
  from invoice as iv
  left join inventory as inv on iv.inventory_id = inv.inventory_id
  left join salespeople as se on iv.salesperson_id = se.salesperson_id;
```

The full details of the relational demonstration, as well as the opportunity to load and run the demonstration, can be gotten from the MySQL code file relational_demo.sql.

Example Database

The MySQL organization offers an interesting, free sample relational database, using methods more advanced than in this tutorial, composed of make-believe movie data, that the reader can use for exploration and harmless experimentation. I say "harmless" because if the user inadvertently deletes or changes something that prevents the database from working, it's a matter of seconds to restore it from source. Here are the steps to get the practice database installed locally:

- Download the MySQL organizations's sample database archive sakila-db.tar.gz from here .
- Unpack the archive:
  - `tar -xzf sakila-db.tar.gz`
- The above command will produce a new directory (sakila-db) containing three files:

    - sakila-db/sakila-data.sql
    - sakila-schema.sql
    - sakila.mwb
- In that new directory, run the mysql command-line interpreter:
    - mysql -p (password)
- Now issue these instructions (green text to the right of the mysql prompt):
    - mysql> source sakila-schema.sql
    - mysql> source sakila-data.sql

If all the above instructions are carried out correctly, a new database named "sakila" will be present in your MySQL database list. The database contains a number of tables and views ("views" are described above in "MySQL Command Summary") typical of a modern database.

The reader should feel free to experiment with this database, be willing to make mistakes, because the database can be easily restored from its source by repeating the above procedure.

Topical Links

- Wikipedia: Relational database
  Wikipedia: Hierarchical database model
- MySQL: FOREIGN KEY Constraints
- MySQL Workbench

— Navigate this multi-page article with the arrows and drop-down lists at the top and bottom of each page —