Databases - Classic Models

Gordon Royle

School of Mathematics & Statistics
University of Western Australia

This lecture

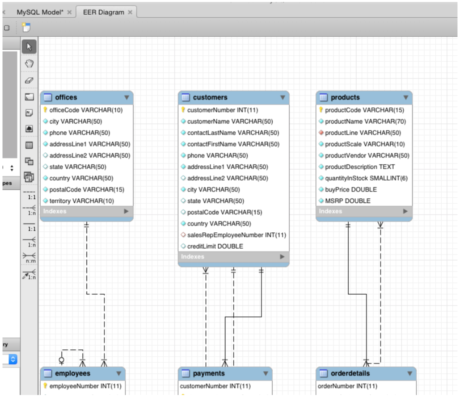
This lecture analyses a freely available “sample database” known as “Classic Models”.

`http://www.mysqltutorial.org/mysql-sample-database.aspx`

This is an 8-table database representing the operations of a business selling *classic models* of cars, trains etc.

Install and open

After the database is installed, we can use the MySQL Workbench to *reverse engineer* it and draw the structure diagram.



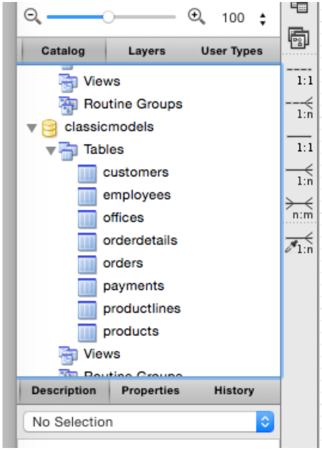
The schema

The diagram shows

- The *tables* in the database
- The *attributes* (that is, columns) of each table
- The *relationships* between the tables
- The *constraints* on the tables

The tables

The database contains 8 tables



Views
Routine Groups
classicmodels
Tables
customers
employees
offices
orderdetails
orders
payments
productlines
products
Views
Routine Groups

A simple table

One way to start understandng a table is to see how it was created.

```
mysql> SHOW CREATE TABLE payments;
CREATE TABLE `payments` (
  `customerNumber` int(11) NOT NULL,
  `checkNumber` varchar(50) NOT NULL,
  `paymentDate` date NOT NULL,
  `amount` double NOT NULL,
  PRIMARY KEY (`customerNumber`,`checkNumber`),
  CONSTRAINT `payments_ibfk_1` FOREIGN KEY (`customerNumber`)
      REFERENCES `customers` (`customerNumber`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Analysis 1

The table has four columns

- The `customerNumber` which is an `INT`
- The `checkNumber` which is a string of characters
  Note that *check* is the US-English version of *cheque*.
- The `paymentDate` which is when the payment was made
  This is of type `DATE` which is a built in SQL type.
- The `amount` which is a double precision floating number
  This allows numbers like 1001.25 to be stored.

Analysis 2

The table has a *primary key*

```
(customerNumber, checkNumber)
```

which means that the *combination* of these values uniquely determines a payment. Declaring this to MySQL has two effects — firstly it prevents double data entry, and secondly it creates an *index* that makes retrieving rows with a particular value of the key very rapid.

In other words, two different rows cannot have the same values for *both* `customerNumber` and `checkNumber`.

Intuitively this means that if you know both the customer number and check number, then you can precisely track down the payment.

Analysis 3

The final feature is

```
CONSTRAINT ‘payments_ibfk_1‘ FOREIGN KEY (‘customerNumber‘)
    REFERENCES ‘customers‘ (‘customerNumber‘)
```

This specifies a *constraint* that must be satisfied by every row of the table.

Don’t worry - some of this ugly-looking constraint has been auto-generated by MySQL, and the syntax that the DBA uses *specifying* a constraint is not as complicated.

Analysis 4

- The constraint is called `payments_ibfk_1` — this name would have been generated automatically
- The constraint is a `FOREIGN KEY` constraint
- The constraint says that the field `customerNumber` (in the table being defined) must be a *reference to* the field `customers(customerNumber)`, which is the `customerNumber` field of the `customers` table.

This *forces* every entry in the `payments` table to refer to a legitimate customer number in the `customer` table.

Orders

The table `orders` also has a field called `customerNumber` and it too is a foreign key to the `customers` table.

A table containing the *references* is called the *child table*, while the table the references *refer to* is called the *parent table*.

So in this case, both `payments` and `orders` are child tables with parent table `customers`.

Customers

```
SELECT customerNumber
FROM   orders;
+----------------+
| customerNumber |
+----------------+
|            103 |
|            103 |
|            103 |
|            112 |
|            112 |
|            112 |
|            114 |
|            114 |
|            114 |
...
```

Deleting customers

Suppose that customer 112 retires and closes up his business, so we want to remove that record.

The syntax for *deleting rows* is to simply replace `SELECT` with `DELETE`.

```
DELETE FROM customers
WHERE  customerNumber = 112;
```

So this command should delete customer 112.

But what happens?

```
mysql> DELETE FROM customers WHERE customerNumber = 112;
ERROR 1451 (23000): Cannot delete or update a parent row:
  a foreign key constraint fails
  (`classicmodels`.`orders`,
    CONSTRAINT `orders_ibfk_1`
    FOREIGN KEY (`customerNumber`)
    REFERENCES `customers` (`customerNumber`))
```

The deletion fails, but with a complicated error message.

Referential Integrity

The error message says that the system *"Cannot delete or update a parent row"*.

The *parent row* is the row in `customers` for customer number 112, so the system is reporting that it cannot delete this row.

It then gives the reason, which is that if the row were to be deleted, then the foreign key constraint whose name is `orders_ibfk_1` would be violated.

The requirement that all references must be valid is called *referential integrity*.

So what now?

In this case, the business must:

- Delete all the *orders* involving customer 112,
- Delete all the *payments* involving customer 112,
- Then the row in `customers` can safely be deleted

If this is a frequent occurrence, then the entire process can be automated by specifying *rules* for what should happen to the child rows when the parent row is changed, as part of the `CREATE TABLE` command for the child table.

Offices

```
DESCRIBE offices;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| officeCode   | varchar(10) | NO   | PRI | NULL    |       |
| city         | varchar(50) | NO   |     | NULL    |       |
| phone        | varchar(50) | NO   |     | NULL    |       |
| addressLine1 | varchar(50) | NO   |     | NULL    |       |
| addressLine2 | varchar(50) | YES  |     | NULL    |       |
| state        | varchar(50) | YES  |     | NULL    |       |
| country      | varchar(50) | NO   |     | NULL    |       |
| postalCode   | varchar(15) | NO   |     | NULL    |       |
| territory    | varchar(10) | NO   |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
9 rows in set (0.01 sec)
```

The `offices` table lists address details for each office and assigns each a unique code.

Offices

```
SELECT officeCode,
       city
FROM   offices;
+------------+---------------+
| officeCode | city          |
+------------+---------------+
| 1          | San Francisco |
| 2          | Boston        |
| 3          | NYC           |
| 4          | Paris         |
| 5          | Tokyo         |
| 6          | Sydney        |
| 7          | London        |
+------------+---------------+
7 rows in set (0.00 sec)
```

Who uses offices?

The structure diagram shows us that the only table *referring to* `offices` is the `employees` table.

This shows that each employee has exactly one office code, hence is attached to exactly one office.

Who’s in Sydney?

What are the names of the Sydney employees?

Who's in Sydney?

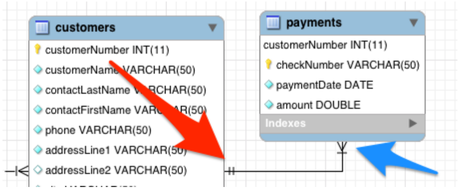
What are the names of the Sydney employees?

```
SELECT E.lastname
FROM   employees E,
       offices O
WHERE  E.officecode = O.officecode
       AND O.city = 'Sydney';
+-----------+
| lastName  |
+-----------+
| Patterson |
| Fixter    |
| Marsh     |
| King      |
+-----------+
```

Who’s in Sydney?

```
SELECT CONCAT(E.firstname, ' ', E.lastname)
FROM   employees E,
       offices O
WHERE  E.officeCode = O.officeCode
       AND O.city = 'Sydney';
+---------------------------------------+
| CONCAT(E.firstName, ' ', E.lastName) |
+---------------------------------------+
| William Patterson                    |
| Andy Fixter                          |
| Peter Marsh                          |
| Tom King                             |
+---------------------------------------+
```

Crow’s foot notation



The double vertical bar (red arrow) means “exactly one” while the crows foot (blue arrow) means “many” (more than one).

These are read in a “look across” fashion, so the diagram shows that each payment is associated with *exactly one* customer, and each customer with *many* payments.

Orders

How are *orders* represented in the DB?

There are two tables dealing with orders

- The table `orders`
  Each row of this table represents one order with a unique *order number* and a foreign key reference to a *customer*.
- The table `orderdetails`
  Each row of this table contains information about *one item* in *one order*, so the whole order consists of *all the rows* of this table with a particular order number.

What’s an order?

```
SELECT orderNumber,
       productCode,
       quantityOrdered AS quant,
       priceEach
FROM   orderdetails
WHERE  orderNumber = 10100;
+-------------+-------------+-------+-----------+
| orderNumber | productCode | quant | priceEach |
+-------------+-------------+-------+-----------+
|       10100 | S18_1749    |    30 |       136 |
|       10100 | S18_2248    |    50 |     55.09 |
|       10100 | S18_4409    |    22 |     75.46 |
|       10100 | S24_3969    |    49 |     35.29 |
+-------------+-------------+-------+-----------+
4 rows in set (0.00 sec)
```

Managing orders

This sort of structure is very common in a database where a “compound object” can have an arbitrary number of parts.

Rather than try to accommodate the *entire order* in a single row, like:

```
(num, item1, quant1, price1, item2, quant2, price2, item3, quant3, ...)
```

the component parts of the order are each stored as separate rows in a table.

This allows an order to contain *any number* of products without specifying how many in advance.

But what is the price?

So how do I find the *price* of an order, if its component parts are stored separately?

The answer is a combination of *joins* and *aggregation*:

```
SELECT orderNumber,
       Sum(quantityOrdered * priceEach) AS totalPrice
FROM   orderdetails
GROUP  BY orderNumber;
+-------------+------------+
| orderNumber | totalPrice |
+-------------+------------+
|       10100 |   10223.83 |
|       10101 |   10549.01 |
|       10102 |    5494.78 |
```

No redundancy principle

One of the major aims of a relational database is to *avoid redundancy* so that each piece of information is stored in *one place* only.

If some combination of information stored in different tables is required, then *joins* are used to combine the tables and extract what is required.

Which orders involve planes?

We want to find out which orders involve *planes*.

```
SELECT DISTINCT orderNumber
FROM   orderdetails
       NATURAL JOIN products
WHERE  productLine = 'Planes';
+-------------+
| orderNumber |
+-------------+
|       10106 |
|       10119 |
|       10131 |
...
```

Reporting

The `employees` table shows us that the data (i.e. row) for each employee refers to *another employee*, who is their immediate supervisor — in other words, the person they *report to*.

Who does 1702 report to?

```
SELECT reportsTo
FROM   employees
WHERE  employeeNumber = 1702;
+-----------+
| reportsTo |
+-----------+
|      1102 |
+-----------+
```

Adding names

What is the *name* of 1702's supervisor?

```
SELECT firstname,
       lastname
FROM   employees
WHERE  employeeNumber = (SELECT reportsTo
                         FROM   employees
                         WHERE  employeeNumber = 1702);
+-----------+----------+
| firstname | lastName |
+-----------+----------+
| Gerard    | Bondur   |
+-----------+----------+
```

Up the hierarchy

```
SELECT reportsTo
FROM   employees
WHERE  employeeNumber = 1102;
+-----------+
| reportsTo |
+-----------+
|      1056 |
+-----------+

SELECT reportsTo
FROM   employees
WHERE  employeeNumber = 1056;
+-----------+
| reportsTo |
+-----------+
|      1002 |
+-----------+

SELECT reportsto
FROM   employees
WHERE  employeenumber = 1002;
+-----------+
| reportsTo |
+-----------+
|      NULL |
+-----------+
```

`NULL`

The “value” `NULL` has a special meaning in SQL and behaves in logical, but sometimes counterintuitive, ways.

It is used as a “placeholder” to represent *“don’t know”* or *“does not apply”*, i.e. missing or inapplicable data.

In this case, the President of the company does not report to anyone, so the value of the `reportsTo` field is set to `NULL`.

Properties of NULL

A NULL entry is not treated as a *value*

```
SELECT firstName,
       lastName,
       jobtTitle
FROM   employees
WHERE  reportsTo = NULL;
Empty set (0.01 sec)
```

but there is a special mechanism for finding it in a table - using IS NULL.

```
SELECT firstName,
       lastName,
       jobtTitle
FROM   employees
WHERE  reportsTo IS NULL;
+-----------+----------+-----------+
| firstName | lastName | jobTitle  |
+-----------+----------+-----------+
| Diane     | Murphy   | President |
+-----------+----------+-----------+
```