A Quick-Start Tutorial on Relational Database Design

Introduction

Relational database was proposed by Edgar Codd (of IBM Research) around 1969. It has since become the dominant database model for commercial applications (in comparison with other database models such as hierarchical, network and object models). Today, there are many commercial *Relational Database Management System* (RDBMS), such as Oracle, IBM DB2 and Microsoft SQL Server. There are also many free and open-source RDBMS, such as MySQL, mSQL (mini-SQL) and the embedded JavaDB (Apache Derby).

A relational database organizes data in *tables* (or *relations*). A table is made up of rows and columns. A row is also called a *record* (or *tuple*). A column is also called a *field* (or *attribute*). A database table is similar to a spreadsheet. However, the relationships that can be created among the tables enable a relational database to efficiently store huge amount of data, and effectively retrieve selected data.

A language called SQL (Structured Query Language) was developed to work with relational databases.

Database Design Objective

A well-designed database shall:

- Eliminate Data Redundancy: the same piece of data shall not be stored in more than one place. This is because duplicate data not only waste storage spaces but also easily lead to inconsistencies.
- Ensure Data Integrity and Accuracy:
- [TODO] more

Relational Database Design Process

Database design is more art than science, as you have to make many decisions. Databases are usually customized to suit a particular application. No two customized applications are alike, and hence, no two database are alike. Guidelines (usually in terms of what not to do instead of what to do) are provided in making these design decision, but the choices ultimately rest on the you - the designer.

Step 1: Define the Purpose of the Database (Requirement Analysis)

Gather the requirements and define the objective of your database, e.g. ...

Drafting out the sample input forms, queries and reports, often helps.

Step 2: Gather Data, Organize in tables and Specify the Primary Keys

Once you have decided on the purpose of the database, gather the data that are needed to be stored in the database. Divide the data into subject-based tables.

Choose one column (or a few columns) as the so-called *primary key*, which uniquely identify the each of the rows.

Primary Key

In the relational model, a table cannot contain duplicate rows, because that would create ambiguities in retrieval. To ensure uniqueness, each table should have a column (or a set of columns), called *primary key*, that uniquely identifies every records of the table. For example, an unique number `customerID` can be used as the primary key for the `Customers` table; `productCode` for `Products` table; `isbn` for `Books` table. A primary key is called a *simple key* if it is a single column; it is called a *composite key* if it is made up of several columns.

Most RDBMSs build an index on the primary key to facilitate fast search and retrieval.

The primary key is also used to reference other tables (to be elaborated later).

You have to decide which column(s) is to be used for primary key. The decision may not be straight forward but the primary key shall have these properties:

- The values of primary key shall be unique (i.e., no duplicate value). For example, `customerName` may not be appropriate to be used as the primary key for the `Customers` table, as there could be two customers with the same name.
- The primary key shall always have a value. In other words, it shall not contain NULL.

Consider the followings in choose the primary key:

- The primary key shall be simple and familiar, e.g., `employeeID` for `employees` table and `isbn` for `books` table.
- The value of the primary key should not change. Primary key is used to reference other tables. If you change its value, you have to change all its references; otherwise, the references will be lost. For example, `phoneNumber` may not be appropriate to be used as primary key for table `Customers`, because it might change.
- Primary key often uses integer (or number) type. But it could also be other types, such as texts. However, it is best to use numeric column as primary key for efficiency.
- Primary key could take an arbitrary number. Most RDBMSs support so-called *auto-increment* (or `AutoNumber` type) for integer primary key, where (current maximum value + 1) is assigned to the new record. This arbitrary number is *fact-less*, as it contains no factual information. Unlike factual information such as phone number, fact-less number is ideal for primary key, as it does not change.
- Primary key is usually a single column (e.g., `customerID` or `productCode`). But it could also make up of several columns. You should use as few columns as possible.

Let's illustrate with an example: a table `customers` contains columns `lastName`, `firstName`, `phoneNumber`, `address`, `city`, `state`, `zipCode`. The candidates for primary key are `name=`

(lastName, firstName), phoneNumber, Address1=(address, city, state), Address1=(address, zipCode). Name may not be unique. Phone number and address may change. Hence, it is better to create a fact-less auto-increment number, say customerID, as the primary key.

Step 3: Create Relationships among Tables

A database consisting of independent and unrelated tables serves little purpose (you may consider to use a spreadsheet instead). The power of relational database lies in the relationship that can be defined between tables. The most crucial aspect in designing a relational database is to identify the relationships among tables. The types of relationship include:

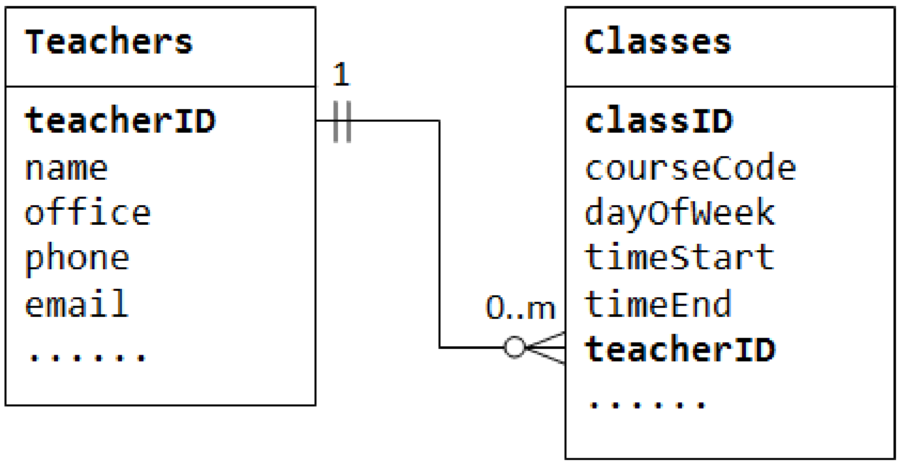
1. one-to-many
2. many-to-many
3. one-to-one

One-to-Many

In a "class roster" database, a teacher may teach zero or more classes, while a class is taught by one (and only one) teacher. In a "company" database, a manager manages zero or more employees, while an employee is managed by one (and only one) manager. In a "product sales" database, a customer may place many orders; while an order is placed by one particular customer. This kind of relationship is known as *one-to-many*.

One-to-many relationship cannot be represented in a single table. For example, in a "class roster" database, we may begin with a table called Teachers, which stores information about teachers (such as name, office, phone and email). To store the classes taught by each teacher, we could create columns class1, class2, class3, but faces a problem immediately on how many columns to create. On the other hand, if we begin with a table called Classes, which stores information about a class (courseCode, dayOfWeek, timeStart and timeEnd); we could create additional columns to store information about the (one) teacher (such as name, office, phone and email). However, since a teacher may teach many classes, its data would be duplicated in many rows in table Classes.

To support a one-to-many relationship, we need to design two tables: a table Classes to store information about the classes with classID as the primary key; and a table Teachers to store information about teachers with teacherID as the primary key. We can then create the one-to-many relationship by storing the primary key of the table Teacher (i.e., teacherID) (the "one"-end or the *parent table*) in the table classes (the "many"-end or the *child table*), as illustrated below.



The column `teacherID` in the child table `Classes` is known as the *foreign key*. A foreign key of a child table is a primary key of a parent table, used to reference the parent table.

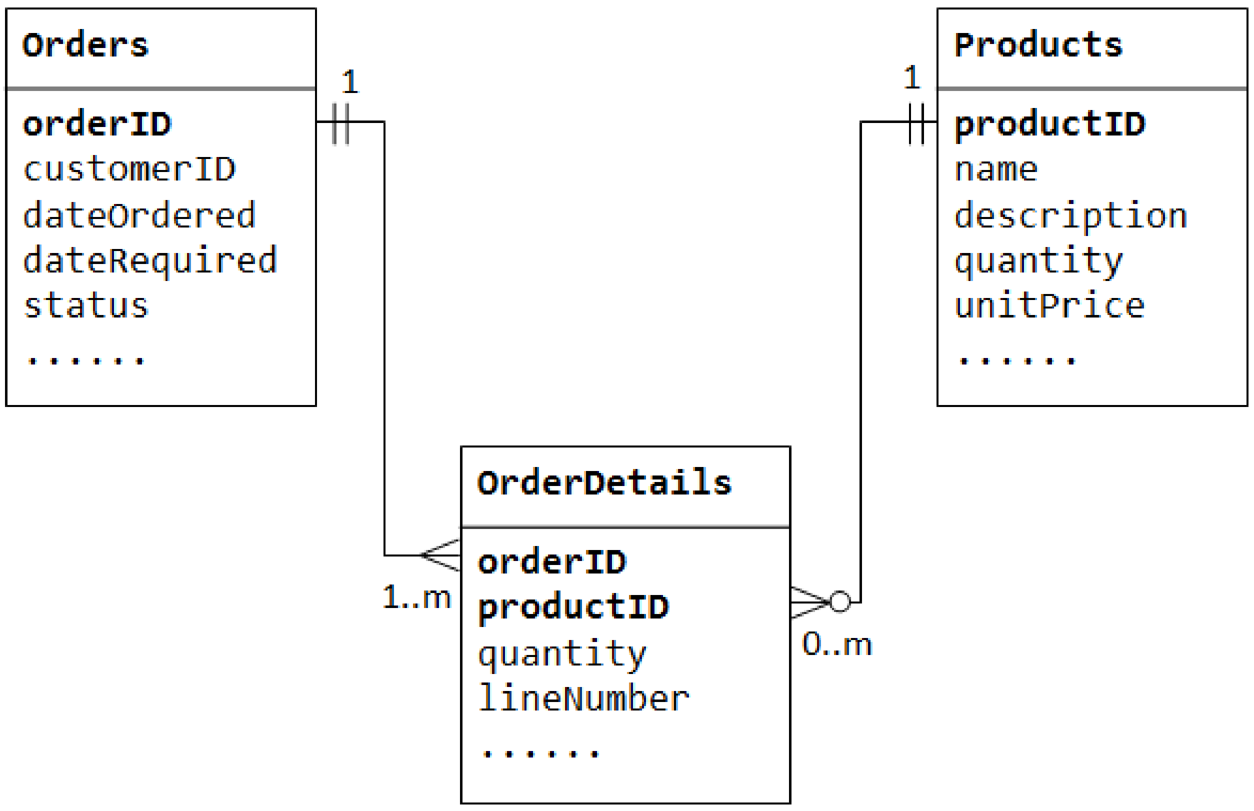
Take note that for every value in the parent table, there could be zero, one, or more rows in the child table. For every value in the child table, there is one and only one row in the parent table.

Many-to-Many

In a "product sales" database, a customer's order may contain one or more products; and a product can appear in many orders. In a "bookstore" database, a book is written by one or more authors; while an author may write zero or more books. This kind of relationship is known as *many-to-many*.

Let's illustrate with a "product sales" database. We begin with two tables: `Products` and `Orders`. The table `products` contains information about the products (such as `name`, `description` and `quantityInStock`) with `productID` as its primary key. The table `orders` contains customer's orders (`customerID`, `dateOrdered`, `dateRequired` and `status`). Again, we cannot store the items ordered inside the `Orders` table, as we do not know how many columns to reserve for the items. We also cannot store the order information in the `Products` table.

To support many-to-many relationship, we need to create a third table (known as a *junction table*), say `OrderDetails` (or `OrderLines`), where each row represents an item of a particular order. For the `OrderDetails` table, the primary key consists of two columns: `orderID` and `productID`, that uniquely identify each row. The columns `orderID` and `productID` in `OrderDetails` table are used to reference `Orders` and `Products` tables, hence, they are also the foreign keys in the `OrderDetails` table.



The many-to-many relationship is, in fact, implemented as two one-to-many relationships, with the introduction of the junction table.

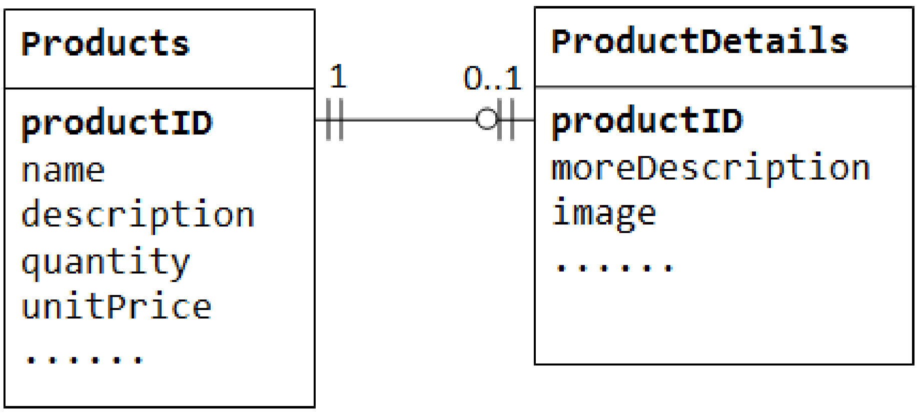
1. An order has many items in `OrderDetails`. An `OrderDetails` item belongs to one particular order.
2. A product may appears in many `OrderDetails`. Each `OrderDetails` item specified one product.

One-to-One

In a "product sales" database, a product may have optional supplementary information such as `image`, `moreDescription` and `comment`. Keeping them inside the `Products` table results in many empty spaces (in those records without these optional data). Furthermore, these large data may degrade the performance of the database.

Instead, we can create another table (say `ProductDetails`, `ProductLines` or `ProductExtras`) to store the optional data. A record will only be created for those products with optional data. The two tables, `Products` and `ProductDetails`, exhibit a *one-to-one relationship*. That is, for every row in the parent table, there is at most one row (possibly zero) in the child table. The same column `productID` should be used as the primary key for both tables.

Some databases limit the number of columns that can be created inside a table. You could use a one-to-one relationship to split the data into two tables. One-to-one relationship is also useful for storing certain sensitive data in a secure table, while the non-sensitive ones in the main table.



Column Data Types

You need to choose an appropriate data type for each column. Commonly data types include: integers, floating-point numbers, string (or text), date/time, binary, collection (such as enumeration and set).

Step 4: Refine & Normalize the Design

For example,

- adding more columns,
- create a new table for optional data using one-to-one relationship,
- split a large table into two smaller tables,
- others.

Normalization

Apply the so-called *normalization rules* to check whether your database is structurally correct and optimal.

First Normal Form (1NF): A table is 1NF if every cell contains a single value, not a list of values. This properties is known as *atomic*. 1NF also prohibits repeating group of columns such as `item1`, `item2`,.., `itemN`. Instead, you should create another table using one-to-many relationship.

Second Normal Form (2NF): A table is 2NF, if it is 1NF and every non-key column is fully dependent on the primary key. Furthermore, if the primary key is made up of several columns, every non-key column shall depend on the entire set and not part of it.

For example, the primary key of the `OrderDetails` table comprising `orderID` and `productID`. If `unitPrice` is dependent only on `productID`, it shall not be kept in the `OrderDetails` table (but in the `Products` table). On the other hand, if the `unitPrice` is dependent on the product as well as the particular order, then it shall be kept in the `OrderDetails` table.

Third Normal Form (3NF): A table is 3NF, if it is 2NF and the non-key columns are independent of each others. In other words, the non-key columns are dependent on primary key, only on the primary key and nothing else. For example, suppose that we have a `Products` table with columns `productID` (primary key), `name` and `unitPrice`. The column `discountRate` shall not belong to `Products` table if it is also dependent on the `unitPrice`, which is not part of the primary key.

Higher Normal Form: 3NF has its inadequacies, which leads to higher Normal form, such as

Boyce/Codd Normal form, Fourth Normal Form (4NF) and Fifth Normal Form (5NF), which is beyond the scope of this tutorial.

At times, you may decide to break some of the normalization rules, for performance reason (e.g., create a column called `totalPrice` in `Orders` table which can be derived from the `orderDetails` records); or because the end-user requested for it. Make sure that you fully aware of it, develop programming logic to handle it, and properly document the decision.

Integrity Rules

You should also apply the integrity rules to check the integrity of your design:

Entity Integrity Rule: The primary key cannot contain NULL. Otherwise, it cannot uniquely identify the row. For composite key made up of several columns, none of the column can contain NULL. Most of the RDBMS check and enforce this rule.

Referential Integrity Rule: Each foreign key value must be matched to a primary key value in the table referenced (or parent table).

- You can insert a row with a foreign key in the child table only if the value exists in the parent table.
- If the value of the key changes in the parent table (e.g., the row updated or deleted), all rows with this foreign key in the child table(s) must be handled accordingly. You could either (a) disallow the changes; (b) cascade the change (or delete the records) in the child tables accordingly; (c) set the key value in the child tables to NULL.

Most RDBMS can be setup to perform the check and ensure the referential integrity, in the specified manner.

Business logic Integrity: Beside the above two general integrity rules, there could be integrity (validation) pertaining to the business logic, e.g., zip code shall be 5-digit within a certain ranges, delivery date and time shall fall in the business hours; quantity ordered shall be equal or less than quantity in stock, etc. These could be carried out in validation rule (for the specific column) or programming logic.

Column Indexing

You could create *index* on selected column(s) to facilitate data searching and retrieval. An index is a structured file that speeds up data access for `SELECT`, but may slow down `INSERT`, `UPDATE`, and `DELETE`. Without an index structure, to process a `SELECT` query with a matching criterion (e.g., `SELECT * FROM Customers WHERE name='Tan Ah Teck'`), the database engine needs to compare every records in the table. A specialized index (e.g., in BTREE structure) could reach the record without comparing every records. However, the index needs to be rebuilt whenever a record is changed, which results in overhead associated with using indexes.

Index can be defined on a single column, a set of columns (called concatenated index), or part of a column (e.g., first 10 characters of a `VARCHAR(100)`) (called partial index) . You could built more than one index in a table. For example, if you often search for a customer using either `customerName` or `phoneNumber`, you could speed up the search by building an index on column `customerName`, as well as `phoneNumber`. Most RDBMS builds index on the primary key automatically.

REFERENCES & RESOURCES

- "Database design basics (Microsoft Access 2007)", available at http://office.microsoft.com/en-us/access/HA012242471033.aspx.
- Paul Litwin, "Fundamentals of Relational Database Design", available at http://www.deeptraining.com/litwin/dbdesign/FundamentalsOfRelationalDatabaseDesign.aspx.
- Codd E. F., "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, vol. 13, issue 6, pp. 377–387, June 1970.

Latest version tested: MySQL 5.5.15
Last modified: September, 2010

Feedback, comments, corrections, and errata can be sent to Chua Hock-Chuan (ehchua@ntu.edu.sg) | HOME