Relational databases “good sense” guidelines

This document aims to provide basic knowledge and understanding of relational databases and data modeling to help users with no background in data modeling to define well structured models.
Topics covered are some keywords definitions, advantages provided by the use of relational databases, tips about data model design and some considerations regarding the degree of normalization.

Introduction

IBM definition of relational database: “A relational database is a type of database that organizes data into rows and columns, which collectively form a table where the data points are related to each other.”
A relational database represents data in tables, which are composed of rows and columns. Each table corresponds to an entity that is a specific object in the modeled system. Rows, also called records (or tuples), hold data that describes a single instance of the entity distinguished by an unique identifier called key. Columns, also called fields or attributes, contain characteristics of the object represented by that row.
Data is structured across multiple tables which can be joined together through primary or foreign keys, unique identifiers linking tables.

Advantages of Relational Databases

- Easy access and analysis of data through join using Structured Query Language (SQL)
- Reduce redundancy which save space and make data more readable
- Enhance scalability of the structure allowing to add columns and add or remove rows
- Enhance data quality and consistency through type validation, structured storing, errors identification
- Enhance reusability providing clear structure and relationship between data, and facilitate collaboration by offering a common representation.

Relational databases “good sense” guidelines

Designing a database

Requirements

- Primary key uniquely identifies each record in a table, it must contain UNIQUE (and non NULL) values.
- Foreign key in one table (child table) refers to a unique field in another table (referenced or parent table).
- Even though foreign keys usually refer to the Primary Key of the parent table they can refer to other fields if they are unique.
- Primary and foreign keys can be composite, i.e. the unique identifier refers to several columns in the table.
- Each table must have a primary key defined
- If a foreign key refers to a composite Primary Key it must be composite too and include all attributes of the Primary Key.

Design steps

In order to define the structure of the database, one needs to clearly identify entities involved (objects included in the dataset), their attributes (characteristic relative to the object), relationship between entities and keys (unique identifiers and foreign keys to include in child table to allow “join” operation with parent entity).
Moreover, relational database models should comply with several normalization forms from 1st normalization form to the 6th normalization form.
In order to define the structure of the database once can follow the steps define below while keeping in mind to reduce redundancy as much as possible:

1. **Identify entities**
   i.e. objects included in the dataset. For instance a database of a delivery company could have several entities like “customers”, “orders” and “products”.
2. **Identify attributes of each entity**
   i.e. characteristics relative to the object. For instance a customer has a name, a telephone number.
3. **Identify relationship**
   i.e. link between entities. For instance each customer could have once or many orders while an order is shipped to one customer.
4. **Assign keys**
   i.e. identify or define a unique identifier for each entity, assign foreign keys to represent relation between entities

The fifth point should be to comply with normalization forms. Normalization forms provide guidelines that prevent several types of errors from occuring. Normalization purpose’s and errors they prevent from are presented below. However they are not detailed because they are considered too complex. Common sense about data modeling is made evident instead. The first normal forms only will be considered here because it defines sort of mandatory rules to comply with.

Relational databases “good sense” guidelines

Normalization purpose’s

Normalization rules bring many advantages.

- Reduce data redundancy. Enhance organization of data making it easier to store, update and read and reduce the risk of propagating inconsistency.
- Enhance data consistency by storing related data in one single table and referencing it in others.
- Enhance flexibility and scalability making the database tolerant to structure changes like adding fields

Indeed, normalization rules have been implemented to prevent three kinds of errors from occuring:

- Insertion anomaly: can’t create an item without NULL until and unless another one has been created ; create high data redundancy by duplicating same information along rows ;
- Update anomaly: one change in a piece of the information requires to update many/all records
- Deletion anomaly: different entities are in the same table without being dependent on each other and deletion of one record leads to a loss of information relative to one of the entities.

First Normal Form:

- All columns in a table have different names.
- All columns hold values of the same type (same kind of information, expressed in the same way).
- Column and row order doesn’t matter.
- Fields are atomic, i.e. do not contain multiple values.

Relational databases “good sense” guidelines

Case study

This case study is used to describe the process of designing a database by following the steps defined above and reducing redundancy. Suppose one has a dataset with records about customers, their personal data, their orders and orders details.

customer_id	order_id	city	zip	customer_name	products	date_and_time
1001	10523	New-York	10003	John Doe	2pc controller ; 5pc nail file	07/08/2024 15:53
1001	10524	New-York	10003	John Doe	1pc backpack	10/08/2024 7:40
1002	10739	San Francisco	94112	Jane Doe	1pc DVD ; 3pc pen	11/08/2024 18:30
1003	10740	San Francisco	94105	Amanda Smith	1pc phone	11/08/2024 18:37

1. *dataset*

Step 1: Identify entities
Based on fields in the dataset, two entities are immediately visible: customers with their personal data and orders with orders details.

Note: *One could already see another entity with city information or products but keep it simple for now.*

Step 2: Identify attributes
Customers have a customer_id, a customer_name and live in a particular city with a zip code. Orders have an order_id, products list and a date_and_time attributes.

customer_id	customer_name	city	zip
1001	John Doe	New-York	10003
1002	Jane Doe	San Francisco	94112
1003	Amanda Smith	San Francisco	94105

2. *customer table*

order_id	**products**	**date_and_time**
10523	2pc controller ; 5pc nail file	07/08/2024 15:53
10524	1pc backpack	10/08/2024 7:40
10739	1pc DVD ; 3pc pen	11/08/2024 18:30
10740	1pc phone	11/08/2024 18:37

3. *orders table*

Considering the first normal form mentioned before, customer_name should be splitted in customer_name and customer_surname so that it does not contain multiple values. “products” field in the orders table should be splitted likewise so that each row contains a single product and the quantity information in a separate field. An updated version of orders table could looks like below:

customer_id	**order_id**	**product_name**	**quantity**	**date_and_time**
1001	10523	controller	2	07/08/2024 15:53
1001	10523	nail file	5	07/08/2024 15:53
1001	10524	backpack	1	10/08/2024 7:40
1002	10739	controller	1	11/08/2024 18:30
1002	10739	pen	3	11/08/2024 18:30
1003	10740	phone	1	11/08/2024 18:37

4. *orders table updated based on first normal form*

However one can note some redundancy both in customer and orders tables. Indeed, in the orders table customer_id, order_id and date_and_time are replicated with the same content per each product purchased in the same order. The same for cities, if one thousand customers live in New-York, New-York will be repeated one thousand times but the zip code is sufficient (a city may have several zip codes but a zip code is associated with one city only).
Moreover, separating cities from customers and products from orders brings another advantage: it prevents the insertion anomaly mentioned before. Indeed, with the current structure, it is not possible to store information about cities before having a customer living in this city and products can not be stored before being purchased by a customer likewise.

Relational databases “good sense” guidelines

Solution:

- Regarding cities:

It is easy to eliminate redundancy by creating an entity “city”.

customer_id	customer_name	customer_surname	zip
1001	John	Doe	10003
1002	Jane	Doe	94112
1003	Amanda	Smith	94105

5. *customer with lower redundancy and updated based on first normal form*

zip	city
10003	New-York
94112	San Francisco
94105	San Francisco
92020	San Diego

6. *cities table*

Note: *There is no customer from San Diego 92020 yet in the customer table but as cities are separated from customers, there is no insertion anomaly.*

- Regarding products:

To solve the insertion anomaly, a products table should be created.

product_id	product_name	unit_price
1	controller	45
2	nail file	5.40
3	backpack	87
4	pen	2.5
5	phone	899
6	sunglasses	227

7. *products table*

Note: *sunglasses have never been purchased but can be stored in the database and more details regarding products can be stored without increasing redundancy.*

However this does not solve the issue of redundancy in the orders table because even storing product_id instead of product_name, there are still three fields over five that are repeated. Thinking of relationships between entities will help to solve this redundancy issue.

Step 3: Identify relationship
Customers can make one to many orders, a purchase is done by 1 and only one customer.
A customer lives in one city, a city can host many customers.
An order contains one to many purchased products and a product can be purchased 0 to many times.

Note: there is a ‘many to many’ relation between products and orders here. This kind of ‘infinite’ relation usually implies creating a so-called “junction table” to link both entities.
Junction tables are typically composed of unique identifiers from both entities. Here the link (junction table) between orders and products remains in the orders details, i.e. products purchased and in what quantity.

As the junction table should be composed of unique identifiers from both entities, let us go to the fourth step.

Step 4: Assign keys
Primary key must allow to uniquely identify each record in a table. Foreign keys are used to ensure consistency when doing join operation between tables. Primary key fields will be represented in yellow, foreign key attributes will be written in italic.

- customer table:

customer_id uniquely identifies each customer.
The zip code field links the customer table with the cities table, it must be set as a foreign key.

customer_id	**customer_name**	**customer_surname**	**zip**
1001	John	Doe	10003
1002	Jane	Doe	94112
1003	Amanda	Smith	94105

8. *customer table with keys defined*

warning: one could think of a composite primary key on both customer_name and customer_surname but when choosing a Primary Key always anticipate potential future issues like a homonym here.

Relational databases “good sense” guidelines

- cities table:

zip code uniquely identifies each record. (Cities does not uniquely identify because one city can have several zip codes)

zip	city
10003	New-York
94112	San Francisco
94105	San Francisco
92020	San Diego

9. *cities table with key*

- products table:

The products table has a product_id that uniquely identifies each record.

product_id	product_name	unit_price
1	controller	45
2	nail file	5.40
3	backpack	87
4	pen	2.5
5	phone	899
6	sunglasses	227

10. *products table with key*

Note: product_name is also a unique identifier and could be used as a primary key but join operation performed on text field usually takes more time and consumes more resources from computational point of view. It is generally better to have a numerical identifier, especially when there is a large amount of data.

- orders table:

As mentioned earlier, junction tables should be composed of identifiers from both of the two tables they are related to. So the junction table “orders_detailss” that join order and products should contain both product_id identifier from products and order_id from orders (because one single order can be uniquely identified by its order_id).
The junction table “orders_detailss” has a composite primary key made of two foreign keys one from order (order_id) and the other one from products (product_id).

order_id	***customer_id***	**date_and_time**
10523	1001	07/08/2024 15:53
10524	1001	10/08/2024 7:40
10739	1002	11/08/2024 18:30
10740	1003	11/08/2024 18:37

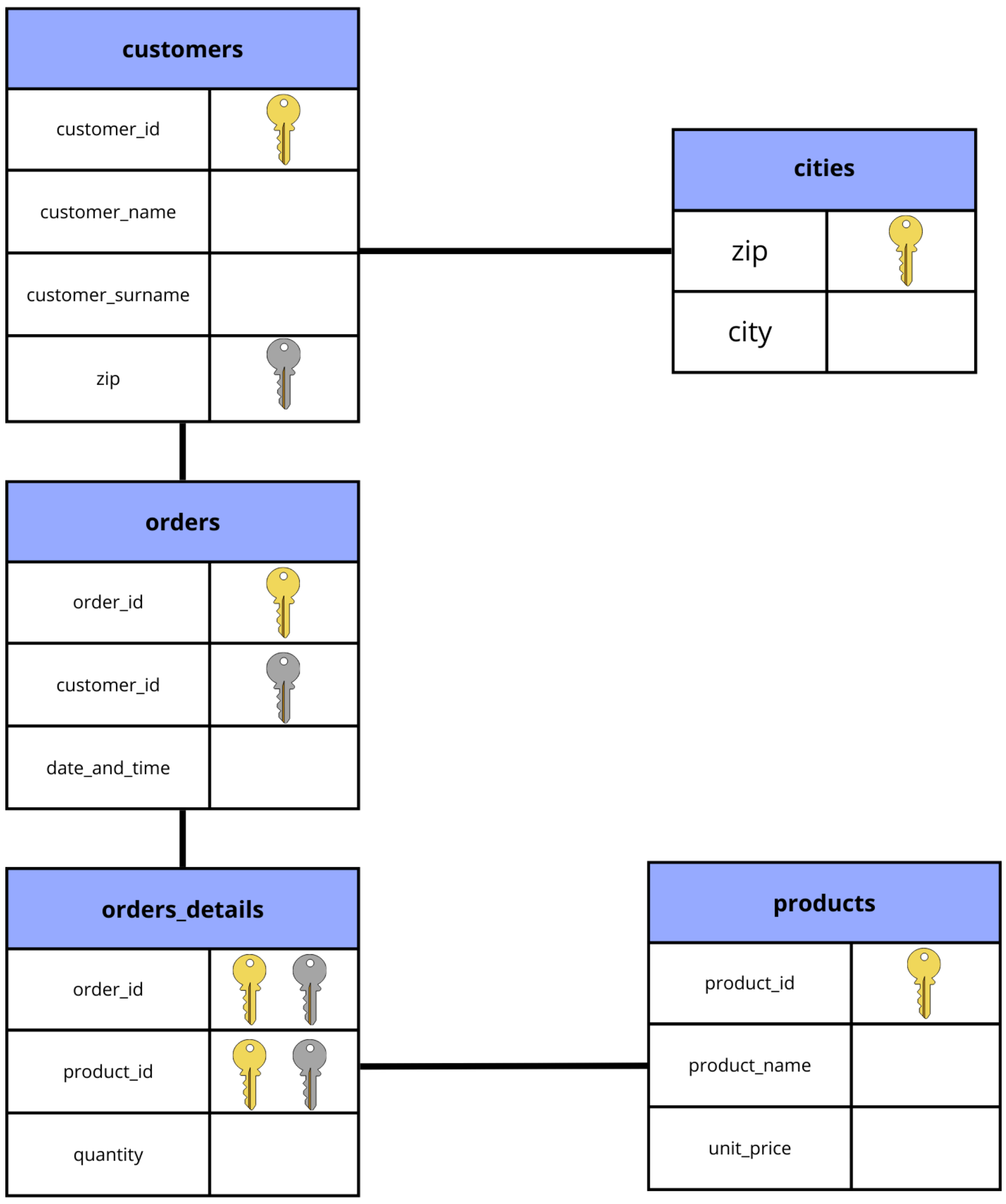
11. *orders table final update*

order_id	***product_id***	**quantity**
10523	1	2
10523	2	5
10524	3	1
10739	1	1
10739	4	3
10740	5	1

12. *orders_details (junction table)*

On the next page an ‘entity-relation’ schema represents the final database with its tables and attributes. It is not a proper entity relationship schema because it does not represent cardinalities between entities.

Relational databases “good sense” guidelines



13. *entity relation like schema*

Relational databases “good sense” guidelines

Degree of normalization

Sort of normalization process has been conducted in the case study without explicitly applying the normalization forms. Those rules ensure a proper logical database structure. However, one should take into consideration the future analysis.
Let us think of the previous case study. If someone want to do some analysis on products bought by all San-Francisco habitants (no matter their zip code) to identify most purchased products for instance, he/she should use this kind of SQL query:

```
SELECT p.product_name,
    SUM(od.quantity) AS total_orders
FROM cities AS c
JOIN customers AS cust ON c.zip = cust.zip
JOIN orders AS o ON cust.customer_id = o.customer_id
JOIN orders_detailss AS od ON o.order_id = od.order_id
JOIN products AS p ON od.product_id = p.product_id
WHERE c.city = 'San Francisco'
GROUP BY p.product_name;
```

This query requires four JOIN operations. JOIN operations can be very resource-intensive and so time consuming on large datasets so one should consider this before going through normalization.
Indeed if a database contains tables with a huge number of rows and that a specific query has large probability to be executed, if this query implies multiple join operation, one could decide to let some redundancy to make this query execute faster.

Relational databases “good sense” guidelines

Sources

IBM - What is a relational database?
IBM - What is data modeling?
Data models for beginners - blog post
A Step-by-Step Database normalization

Going Further

Resources on data models

MariaDB - Database Design

Resources on entity relationship diagram and cardinality

Lucidchart - What is an Entity Relationship Diagram (ERD)?

Resources on normalization

StudyTonight blog - database normalization
MariaDB - Database Normalization
Wikipedia - Database normalization