23 Database Design Principles

Database design principles are essential to creating efficient, reliable, and scalable databases. A database created following these fundamental design principles ensures that its data will be stored in it in an organized and structured manner. It will facilitate database administration and allow users to obtain accurate results.

Following the best practices for data modeling allows a database to meet a series of key objectives:

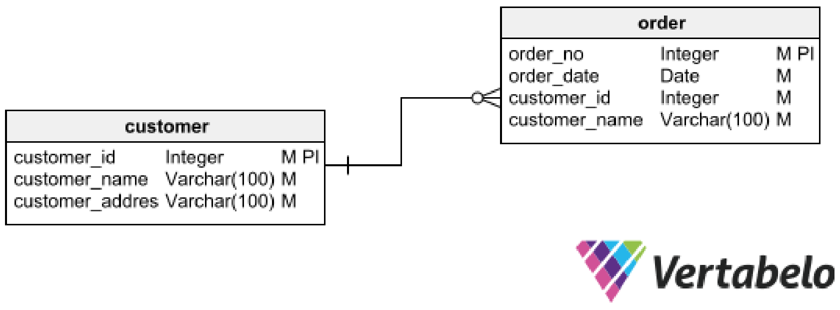
- **Integrity:** Observing principles such as non-redundancy and the use of constraints, primary and foreign keys, data validation, and referential integrity ensures that the information stored in the database maintains its integrity and consistency.
- **Performance:** Normalization principles optimize data access and eliminate redundancy. In turn, indexing principles contribute to speeding up result times.
- **Scalability:** A database must be able to grow without compromising its performance and the integrity of its data. The same design principles that contribute to integrity and performance also make scalability possible.
- **Security:** Database design principles aimed at controlling access to information – as well as encrypting and protecting sensitive data – make information security possible.
- **Maintainability:** Using naming conventions and maintaining up-to-date documentation ensure that databases are easy to use, update, and modify.

12 Key Principles of Database Design

#1: Avoid Redundancy

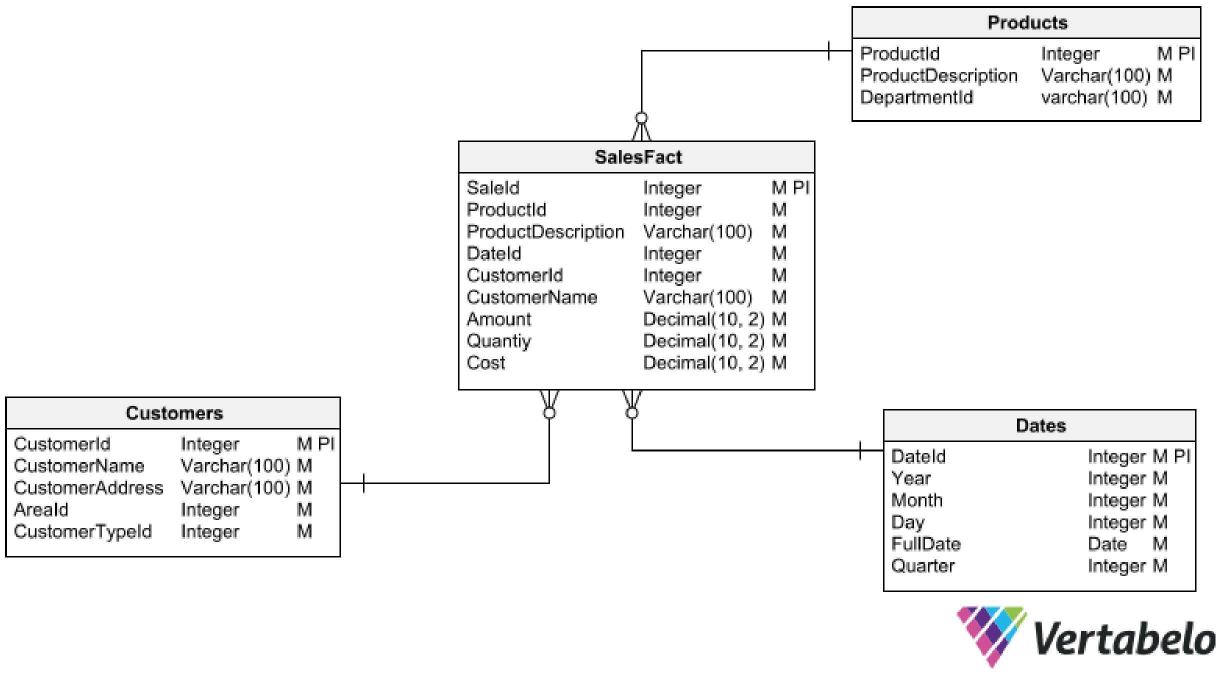
Redundant information in a database schema can cause several problems. It can lead to inconsistencies: if the same data is stored in multiple tables, there is a risk that it will be updated in one table and not in the others – generating discrepancies in the results.

Redundancy also unnecessarily increases the storage space required by the database and can negatively affect query performance and data update operations. The normalization principle (see below) is used to eliminate redundancy in database models.



Above, we see an example of unwanted redundancy in a data model: the `customer_name` column (dependent on `customer_id`) is in both the `customer` table and the `order` table. This creates the risk that it contains different information in each table.

An exception to the principle of non-redundancy occurs in dimensional schemas, which are used for analytical processing and data warehousing. In dimensional schemas, a certain degree of data redundancy can be used to reduce complexity in combined queries. Otherwise, such queries would be more resource intensive.



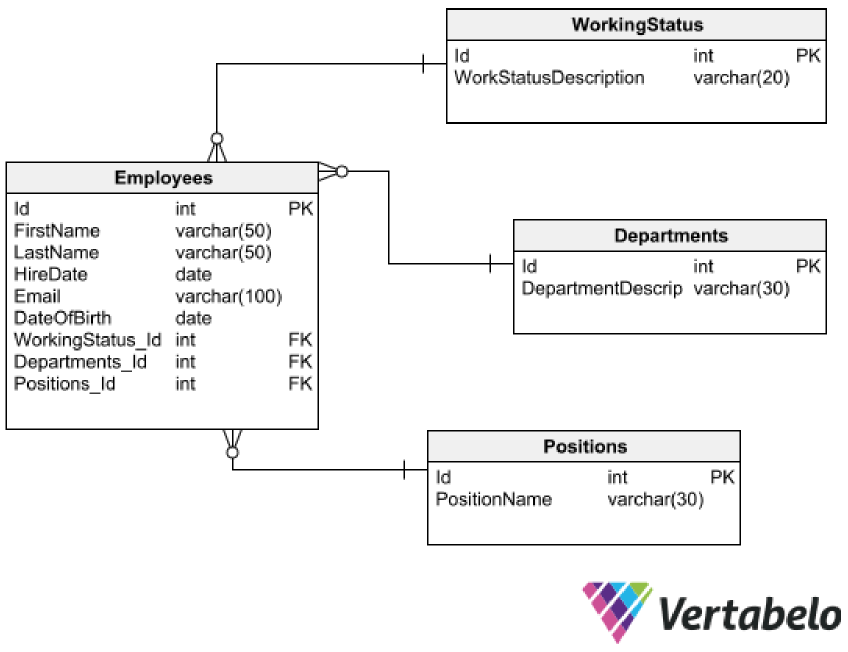
Above we have an example of a dimensional schema. The columns `CustomerName` and `ProductDescription` are repeated in the `SalesFact` table to avoid the need to join tables when querying these columns.

#2: Primary Keys and Unique Identifiers

Every table must have a primary key. The primary key is essential because it guarantees the uniqueness of each row within the table. In addition, the primary key is used to establish relationships with other tables in the database.

Without a primary key, a table would have no reliable way to identify individual records. This can lead to data integrity problems, issues with query accuracy, and difficulties in updating that table. If you leave a table without a primary key in your schema, you run the risk of that table containing duplicate rows that will cause incorrect query results and application errors.

On the other hand, primary keys make it easier to interpret your data model. By seeing the primary keys of every table in an entity-relationship diagram (ERD), the programmer writing a query will know how to access each table and how to join it with others.



Having a primary key in each table ensures that relationships can be maintained between tables.

#3: Null Values

In relational databases, null values indicate unknown, missing, or non-applicable data. When defining each column in a table, you must establish whether it supports null values. You should only allow the column to support null values if you're certain that, at some point, the value of that column may be unknown, missing, or not applicable.

It is also important to differentiate null values from "empty" values, such as the number zero or a zero-length string. Read these tips on how to make good use of nullable columns for more information.

Null values have certain peculiarities:

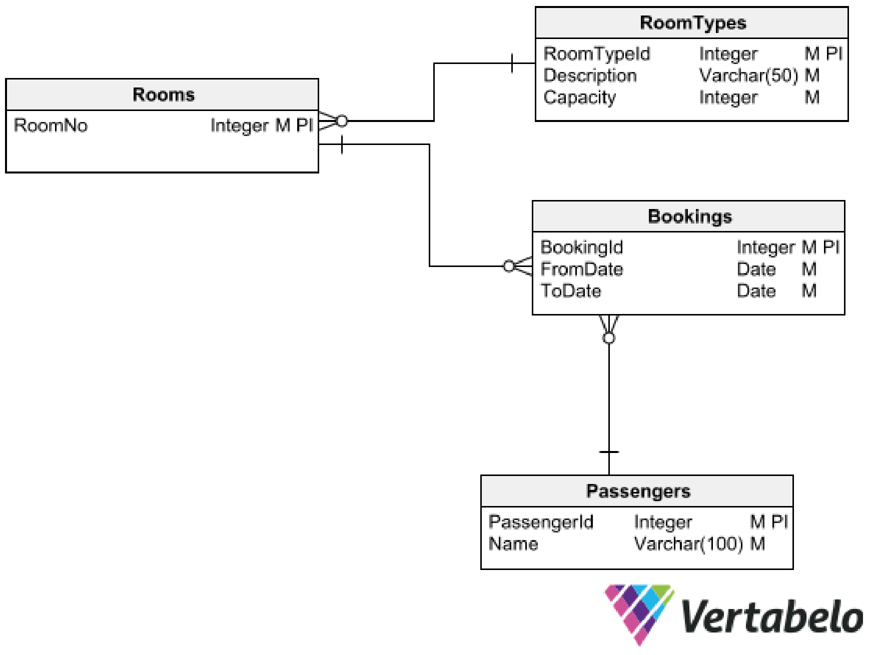
- Primary keys can never store null values.
- A null value can be applied to columns of any data type (as long as the column supports null values).
- Null values are ignored in unique and foreign key constraints.
- In SQL, a null value is different from any value – even from another null value.
- Any SQL operation involving a null value will result in another null value. The exception is mathematical aggregate functions like SUM(), where null values are treated as zeros.

#4: Referential Integrity

Referential integrity guarantees that columns involved in the relationship between two tables (e.g. primary and foreign keys) will have shared values. This means that the values of the columns in the secondary (child or dependent) table must exist in the corresponding columns of the primary (parent) table.

Furthermore, referential integrity requires that such column values are unique in the primary table – ideally, they should constitute the primary key of the primary table. In the secondary table, the columns involved in the relationship must constitute a foreign key. Read What Is a Foreign Key? for more information.

By establishing relationships between tables using foreign keys and primary keys, we make use of the database engine's resources to ensure data integrity. This also improves the performance of queries and other database operations. Foreign and primary keys facilitate the creation of indexes to speed up table lookups; we'll discuss this more in the indexing section of this article.



In this example, referential integrity ensures that each booking is associated with a passenger and a room. It also ensures a room type is specified for each room.

#5: Atomicity

Sometimes, you may feel tempted to have a single column for complex or compound data. For example, the table below stores complete addresses and full names in single fields:

customer_no	customer_name	customer_address
1001	Kaelyn Hodge	5331 Rexford Court, Montgomery AL 36116
2050	Brayden Huang	4001 Anderson Road, Nashville TN 37217
4105	Louis Kelly	2325 Eastridge Circle, Moore OK 73160
3412	Jamarion Dawson	4016 Doane Street, Fremont CA 94538

In this example, the `customer_address` column clearly violates the atomicity principle, since it stores composite data that could be divided into smaller pieces. The `customer_name` field also violates this principle, as it stores first and last names in the same value.

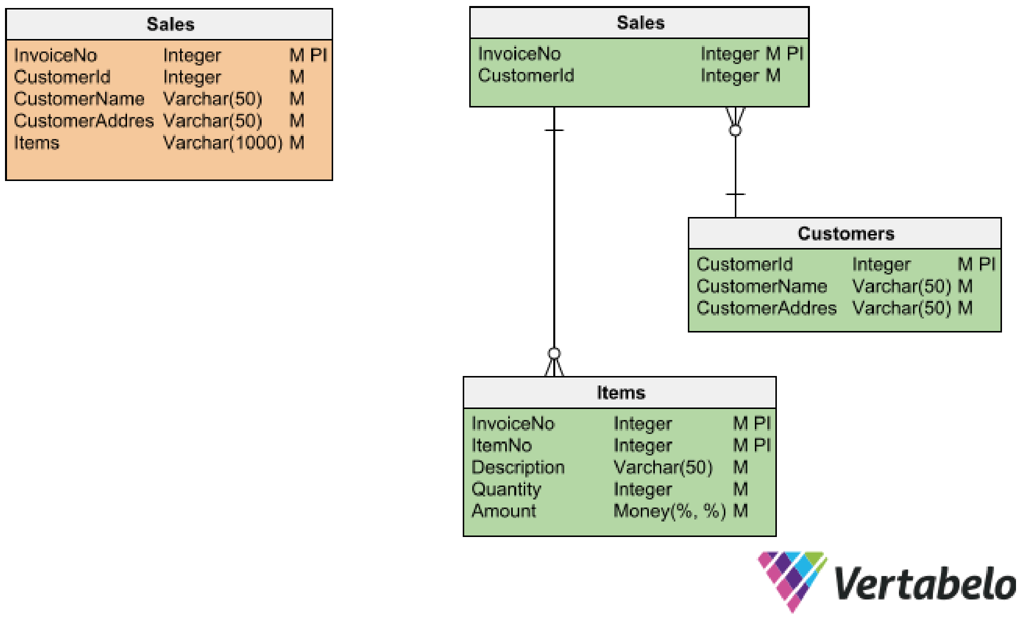
If you combine more than one piece of information in one column, it will be difficult to access individual data later. For example, what if you wanted to address the customer by their first name or see which customers live in the state of Oklahoma? The query would get very complicated!

Try to divide the information into logical parts; in our example, you could create separate fields for first name and last name and for address, city, state, and postal code. This principle is made explicit in the first normal form, which we'll discuss next.

#6: Normalization

In any data model for transactional applications or processes – such as an online banking or e-commerce site – it is crucial to avoid anomalies in the processes of inserting, updating, or deleting data.

This is where normalization techniques are applied; they seek to eliminate clutter, disorganization, and inconsistencies in data models. Normalization is a formal method for correcting data models that you can intuitively sense, just by looking at them, that there is something wrong. Below, the orange table is an example of a non-normalized model:



Any experienced database designer will quickly observe that the orange `Sales` table has many defects that could be solved by normalization. This would result in the green (normalized) tables on the right.

In practice, normalizing a model is a matter of bringing it to the third normal form (3NF). Bringing a data model to 3NF maintains data integrity, reduces redundancy, and optimizes the storage space occupied by the database. The lower normal forms (1NF and 2NF) are intermediate steps towards 3NF, while the higher normal forms (4NF and 5NF) are rarely used. Read our article on normalization and the three normal forms for more information.

#7: Data Typing

When designing a schema, you must choose the appropriate data type for each column of each table. You'll choose a data type according to the nature and format of the information expected to be stored in that column.

If, for example, you are creating a column where telephone numbers will be stored, you could associate a numeric data type to it (such as INT) if it will only store numbers. But, if it must also store other characters – such as parentheses, hyphens, or spaces – the data type should be VARCHAR.

On the other hand, if you create a column to store dates, common database modeling tips suggest that the data type should be DATE, DATETIME, or TIMESTAMP. If you define this column as VARCHAR, dates can be stored in very different formats (e.g. '18-Jan-2023', '2023-01-18', '01/18/23'); it will be impossible to use this column as a search criterion or as a filter for queries.

person	birthdate
Kaelyn Hodge	May 14, 1977
Brayden Huang	2001-06-18
Louis Kelly	12/20/1987
Jamarion Dawson	24 January, 2004

An example of a VARCHAR column used to store dates.

On the other hand, the data type of a column must support the entire universe of possible data that can be assigned to that column, either by users or by applications.

If you create a column to store product prices, for example, you must make sure that it supports the maximum number of integers and decimals with which those prices can be expressed. There are no approximate sizes in this. If you define the data type with less capacity than necessary, you will cause errors in the use of the database. And if you define it with more capacity than necessary, you will be wasting storage space and opening the door to performance problems.

#8: Indexing

Indexes are data structures that make it easy for the database engine to locate the row(s) that meet a given search criteria. Each index is associated with a table and contains the values of one or more columns of that table. Read our article What Is a Database Index? for more information.

Since indexes are created mainly to speed up searches and queries, we don't always know which indexes should be created during the database design process. This is why index creation is often part of database maintenance rather than design.

However, database designers can anticipate the needs of the applications that will access the database and include indexes that promise the greatest benefits. To determine what those indexes are, you can follow these data modeling best practices:

- **Create an index for all tables' primary keys.** This is usually done automatically by relational database management systems (RDBMSs), so you don't usually need to go to the trouble of doing it yourself.
- **Create an index for each foreign key constraint** that exists in the model of the secondary (child) table of each relationship. Some RDBMSs also do this automatically.
- **Create indexes based on use cases.** Analyze how the applications will use the database to anticipate the need for indexes that include the lookup columns detailed in those cases.
- **Create indexes for fields that are frequently used for querying or filtering data**, but avoid creating unnecessary or duplicate indexes.
- **Consider columns' cardinality before you index.** If you create an index for a column with low cardinality, it will rarely speed up queries. This is because it will not significantly reduce the query's search space.
- **Prioritize "narrow" columns for indexes**. Indexes take up storage space, so including very wide columns in an index (such as complete addresses, full names, product descriptions, etc.) will cause the index to take up a lot of space.

These rules are just some starting points for creating indexes. Even when your model design is finished and the database is up and running, you (or a colleague) may still need to create indexes to help maintain optimal performance.

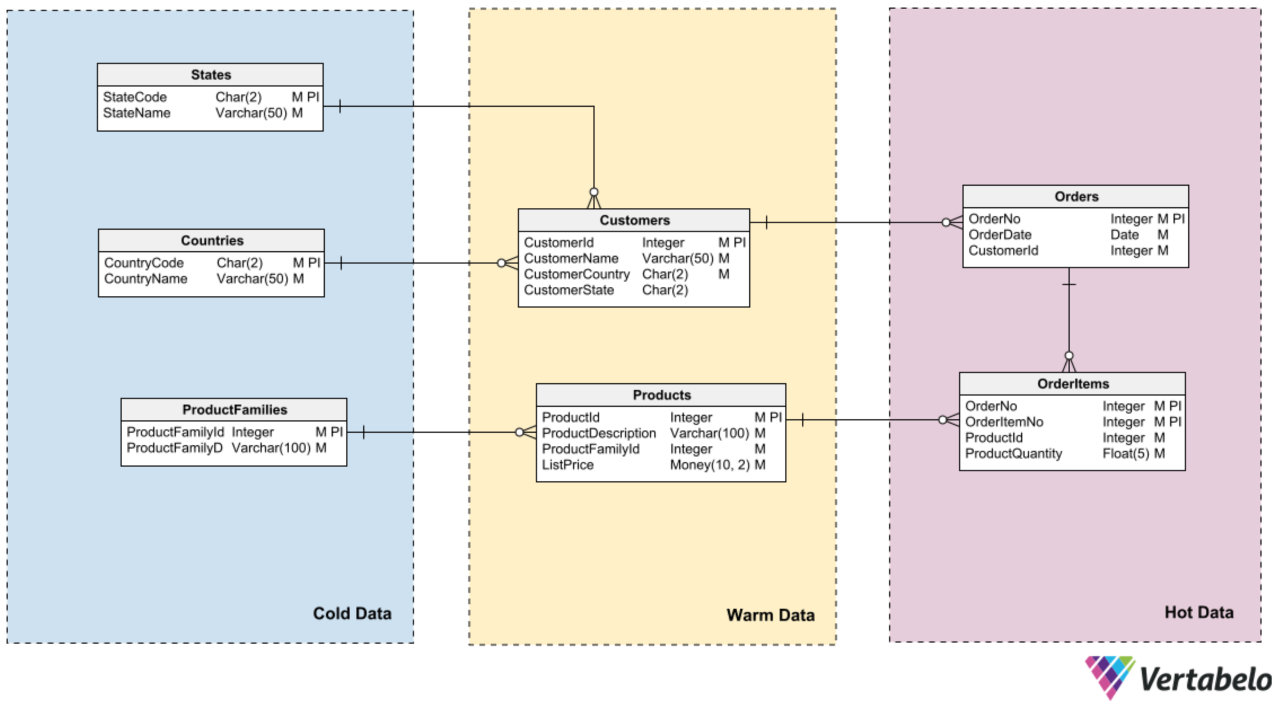
#9: Schema Partitioning

Large schemas are difficult to read and manage when the totality of their tables exceeds the dimensions of a medium-sized poster or a couple of screens. At this point, partitioning the schema becomes necessary so that the schema can be visualized by sections.

The criterion used to partition a large schema is up to the designer and the consumers of that schema (developers, DBAs, database architects, etc.). The idea is to choose the partitioning criteria that is most useful.

For example, a schema can be partitioned according to access permissions. In this case, we'd separate tables with limited read/write permissions from those with free access for any user. Or you might partition a model by update frequency. This calls for separating tables that are updated constantly, tables that are updated occasionally, and tables that are never updated.

These ways of partitioning will be helpful both for DBAs (who need to manage storage assignment, roles, and permissions) and developers (who need to know what kind of interaction applications can have with each table).



A schema that's partitioned according to the update frequency of each table.

#10: Authentication and Access Control

Controlling access to a system through user authentication is one of the most basic principles to prevent data misuse and promote information security.

User authentication is very familiar to all of us; it's how a system, database, program, etc. ensures that 1) a user is who they say they are, and 2) the user only accesses the information or parts of the system they are entitled to see.

A schema intended to provide authentication and access control must allow for registering new users, supporting different authentication factors, and providing options for recovering passwords. It must also protect authentication data from unauthorized users and define user permissions by roles and levels.

Ideally, every schema should support these functionalities. Even if you are asked to design a database without support for security mechanisms – because, for example, it will be used by only one person – it is highly recommended that you include everything necessary to manage access restrictions. If you are asked to leave aside any form of access control to reduce costs or development time, you must make sure that the project stakeholders understand that the design you'll create will be vulnerable to attackers.

For more details on how to create a data model that supports authentication and access control, read these Best Practices for Designing an Authentication Module.

#11: Conceal Sensitive Information

Every database must be prepared to resist hacking and attempts to access data by unauthorized users. Even if security mechanisms such as firewalls and password policies are in place, database design is the last line of defense to protect information when all other methods fail.

There are several things that you, as a designer, can do to minimize the risks of unauthorized access to information. One of them is to provide columns that support encrypted or hashed data. String encryption and hashing techniques alter the length of character strings and the set of characters that can be allowed. When you're defining VARCHAR columns to store data that can be encrypted or hashed, you must take into account both the maximum length and the range of characters they can have.

A common belief is that deliberately using non-obvious names for objects in a database is an effective method of preventing unauthorized users from gaining access to the data. This practice (known as security by obscurity), can make an attacker's "job" somewhat more difficult. But those who know cybersecurity do not recommend this; for an experienced cybercriminal, it only represents a small obstacle to overcome.

When it comes to naming objects in a database, it is essential to have a naming convention for the database objects and to respect it so that no table or column will be named randomly.

#12: Don't Store Authentication Keys

Good design practices for security and user authentication include not storing keys, even encrypted ones. All encrypted data carries the risk of being decrypted. For this reason, hash functions that are not bijective are used to protect keys. This means that there is no way to use a hash function result to obtain the original data. Instead of storing the encrypted key, only the hash of that key is stored.

A hashed key, even if it does not allow finding the original key, serves as an authentication mechanism: if the hash of a password entered during a login session matches the hash stored for the user trying to log in, then there is no doubt that the password entered is the correct one.

It is important to restrict write permissions for the table and column where a hashed password is stored. This helps prevent potential attackers from altering the stored hash to one that corresponds to a known password.