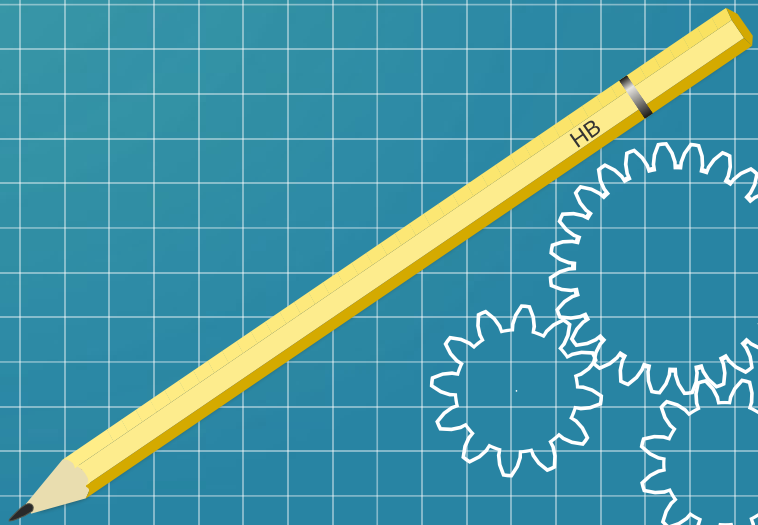
An Introduction to Relational Databases



Introduction

What is a database?

In computing, a database is an organized collection of data or a type of data store based on the use of a database management system (DBMS).

The DBMS additionally encompasses the core facilities provided to administer the database. The sum total of the database, the DBMS and the associated applications can be referred to as a database system.

History

The history of databases can be traced back to the early 1960s when the need for organized data storage and retrieval became evident with the rise of computers and their use in business and scientific applications. Early systems used file structures and hierarchical or network models before the introduction of the relational model in the 1970s. Relational databases, and their query language SQL, have become dominant, but the field continues to evolve with the emergence of NoSQL and other database technologies.

History

1960s: Early Forms and Models

File-Server Systems:

The first computerized databases were based on simple file structures, limited in their ability to efficiently search and process information.

Hierarchical and Network Models:

Two prominent database models emerged: IBM's IMS, a hierarchical model, and CODASYL's approach, a network model, designed to handle complex data relationships.

Apollo Mission:

The need for a centralized system to manage the vast amount of data required for the Apollo program spurred early development in database management.

History

1970s: The Relational Revolution

Codd's Relational Model:

IBM's Edgar F. Codd published a paper in 1970 proposing the relational model, which revolutionized how data was organized and queried using tables.

System R and Ingres:

Two prototype relational database systems, System R and Ingres, were developed in the mid-1970s, laying the groundwork for future relational database systems.

1980s: Commercial Relational Databases

Oracle and DB2: The first commercial relational database systems, including Oracle and IBM's DB2, began to appear in the early 1980s.

History

1990s-Present: Evolving Landscape

SQL and RDBMS Dominance:

Relational databases and their query language SQL became the dominant approach for managing structured data.

NoSQL Rise:

As new needs emerged for managing large amounts of unstructured data, NoSQL databases gained popularity, offering flexible data models and scalability.

Cloud and Serverless Databases:

Cloud-based and serverless database solutions continue to shape the future of database management, offering greater scalability and ease of use.

History

Key Developments:

The development of direct-access storage devices (disks and drums) in the mid-1960s allowed for interactive database access, contrasting with previous batch processing methods .

The introduction of the CODASYL standard in 1971, based on the network model, was a significant step in standardizing database management .

E.F. Codd's relational model, published in 1970, fundamentally changed how people thought about organizing and managing data in databases .

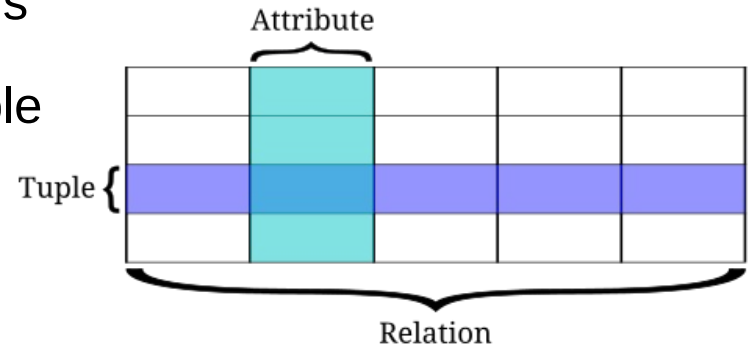
The commercialization of relational database systems in the 1980s, such as Oracle and DB2, marked a major milestone in the adoption of relational databases .

The emergence of NoSQL databases in recent years reflects the need for flexible and scalable solutions for handling large amounts of unstructured data .

Relational model

A relational model organizes data into one or more tables (or "relations") of columns and rows, with a unique key identifying each row. Rows are also called records or tuples. Columns are also called attributes. Generally, each table/relation represents one "entity type" (such as customer or product). The rows represent instances of that type of entity (such as "Lee" or "chair") and the columns represent values attributed to that instance (such as address or price).

For example, each row of a class table corresponds to a class, and a class corresponds to multiple students, so the relationship between the class table and the student table is "one to many"



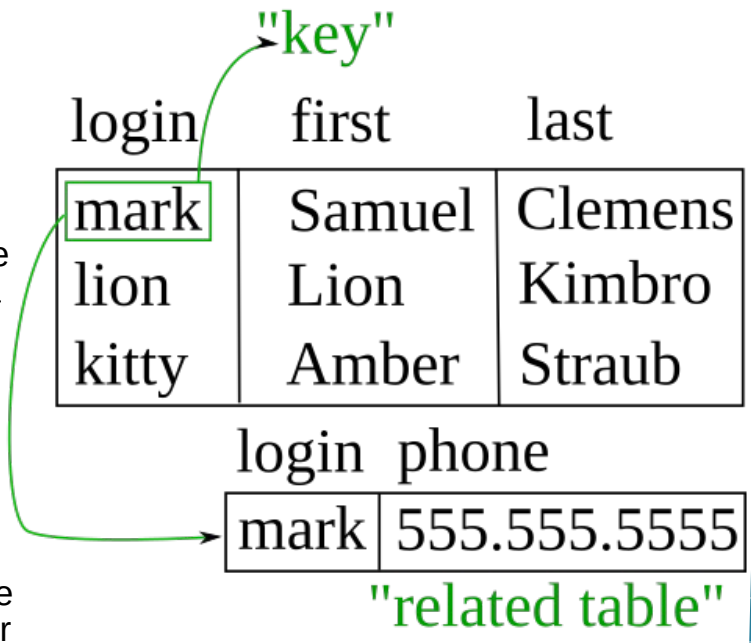
Relational model

Keys

Each row in a table has its own unique key. Rows in a table can be linked to rows in other tables by adding a column for the unique key of the linked row (such columns are known as foreign keys). Codd showed that data relationships of arbitrary complexity can be represented by a simple set of concepts.

Part of this processing involves consistently being able to select or modify one and only one row in a table. Therefore, most physical implementations have a unique primary key (PK) for each row in a table. When a new row is written to the table, a new unique value for the primary key is generated; this is the key that the system uses primarily for accessing the table.

The primary keys within a database are used to define the relationships among the tables. When a PK migrates to another table, it becomes a foreign key (FK) in the other table. When each cell can contain only one value and the PK migrates into a regular entity table, this design pattern can represent either a one-to-one or one-to-many relationship.



Relational operations

Users (or programs) request data from a relational database by sending it a **query**. In response to a **query**, the database returns a result set.

Often, data from multiple tables are combined into one, by doing a **join**. Conceptually, this is done by taking all possible combinations of rows, and then filtering out everything except the answer.

There are a number of relational operations in addition to **join**. These include **project** (the process of eliminating some of the columns), **restrict** (the process of eliminating some of the rows), **union** (a way of combining two tables with similar structures), **difference** (that lists the rows in one table that are not found in the other), **intersect** (that lists the rows found in both tables), and **product** (combines each row of one table with each row of the other).

The flexibility of relational databases allows programmers to write queries that were not anticipated by the database designers. As a result, relational databases can be used by multiple applications in ways the original designers did not foresee, which is especially important for databases that might be used for a long time (perhaps several decades). This has made the idea and implementation of relational databases very popular with businesses.

Relational operations

Terminology

The table below summarizes some of the most important relational database terms and the corresponding SQL term:

SQL term	Relational database term	Description
Row	Tuple or record	A data set representing a single item
Column	***Attribute*** or ***field***	A labeled element of a tuple, e.g. "Address" or "Date of birth"
Table	Relation or Base relvar	A set of tuples sharing the same attributes; a set of columns and rows
View or result set	***Derived relvar***	Any set of tuples; a data report from the RDBMS in response to a query

Relational operations

Relations or tables

In a relational database, a relation is a set of **tuples**, or **rows**, that have the same **attributes,** or **columns**. A row usually represents an object and information about that object. Objects are typically physical objects or concepts. A relation is usually described as a table, **which is organized into rows and columns**. All the data referenced by an attribute are in the same domain and conform to the same constraints.

The relational model specifies that the rows of a relation have no specific order and that the rows, in turn, impose no order on the attributes (columns). Applications access data by specifying queries, which use operations such as select to identify rows, project to identify attributes, and join to combine relations. Relations can be modified using the insert, delete, and update operators. New rows can supply explicit values or be derived from a query. Similarly, queries identify rows for updating or deleting.

Rows by definition are unique. If the row contains a candidate or primary key then obviously it is unique; however, a primary key need not be defined for a row or record to be a row. The definition of a row requires that it be unique, but does not require a primary key to be defined. Because a row is unique, its attributes by definition constitute a superkey.

Relational operations

Constraints

Constraints are often used to make it possible to further restrict the domain of an attribute. For instance, a constraint can restrict a given integer attribute to values between 1 and 10. Constraints provide one method of implementing business rules in the database and support subsequent data use within the application layer. SQL implements constraint functionality in the form of check constraints. Constraints restrict the data that can be stored in relations. These are usually defined using expressions that result in a Boolean value, indicating whether or not the data satisfies the constraint. Constraints can apply to single attributes, to a row (restricting combinations of attributes) or to an entire relation. Since every attribute has an associated domain, there are constraints (domain constraints).

Relational operations

Primary key

Every relation/table has a primary key, this being a consequence of a relation being a set. A primary key uniquely specifies a row within a table. While natural attributes (attributes used to describe the data being entered) are sometimes good primary keys, **surrogate keys** are often used instead. A **surrogate key** is an artificial attribute assigned to an object which uniquely identifies it (for instance, in a table of information about students at a school they might all be assigned a student ID in order to differentiate them). The surrogate key has no intrinsic (inherent) meaning, but rather is useful through its ability to uniquely identify a row. Another common occurrence, especially in regard to N:M cardinality is the composite key. A composite key is a key made up of two or more attributes within a table that (together) uniquely identify a record.

Relational operations

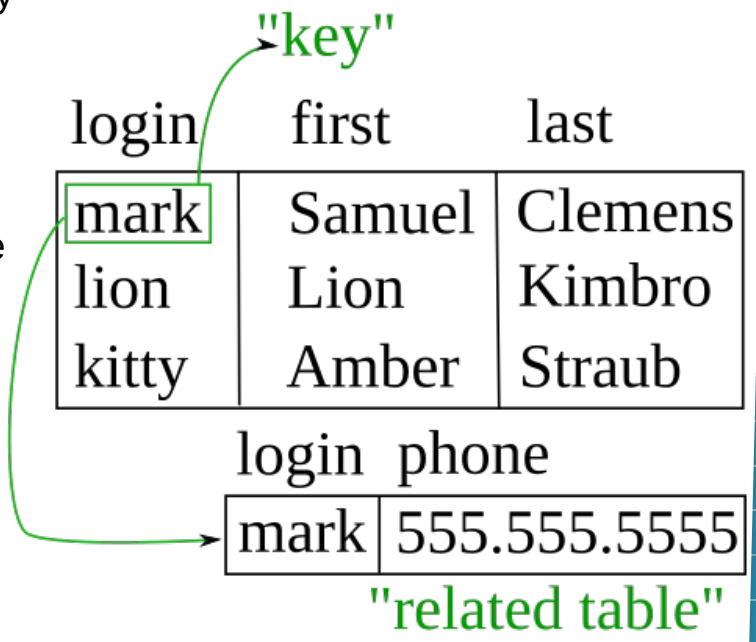
Foreign key

Foreign key refers to a field in a relational table that matches the primary key column of another table. It relates the two keys. Foreign keys need not have unique values in the referencing relation. A foreign key can be used to cross-reference tables, and it effectively uses the values of attributes in the referenced relation to restrict the domain of one or more attributes in the referencing relation. The concept is described formally as: "For all rows in the referencing relation projected over the referencing attributes, there must exist a row in the referenced relation projected over those same attributes such that the values in each of the referencing attributes match the corresponding values in the referenced attributes."

Relational operations

Foreign key

Foreign key refers to a field in a relational table that matches the primary key column of another table. It relates the two keys. Foreign keys need not have unique values in the referencing relation. A foreign key can be used to cross-reference tables, and it effectively uses the values of attributes in the referenced relation to restrict the domain of one or more attributes in the referencing relation. The concept is described formally as: "For all rows in the referencing relation projected over the referencing attributes, there must exist a row in the referenced relation projected over those same attributes such that the values in each of the referencing attributes match the corresponding values in the referenced attributes."



Relational operations

Index

An index is one way of providing quicker access to data. Indices can be created on any combination of attributes on a relation. Queries that filter using those attributes can find matching rows directly using the index (similar to Hash table lookup), without having to check each row in turn. This is analogous to using the index of a book to go directly to the page on which the information you are looking for is found, so that you do not have to read the entire book to find what you are looking for. Relational databases typically supply multiple indexing techniques, each of which is optimal for some combination of data distribution, relation size, and typical access pattern. Indices are usually implemented via B+ trees, R-trees, and bitmaps. Indices are usually not considered part of the database, as they are considered an implementation detail, though indices are usually maintained by the same group that maintains the other parts of the database. The use of efficient indexes on both primary and foreign keys can dramatically improve query performance.

List of database engines

According to DB-Engines, in December 2024 the most popular relational satabases on the db-engines.com web site were:

- Oracle RDBMS
- MySQL/MariaDB
- Microsoft SQL Server
- PostgreSQL
- Snowflake
- IBM Db2
- SQLite
- Microsoft Access
- Databricks
- SAP Sybase
- SQLite

Why use Relational Databases in research computing?

- Your application relies on it.
- It tends to be more structured than using a file system.
- It is somewhat self-documenting.
- Relative easy and efficiency of individual data entry, updates and deletions, retrieval and summarization, e.g.

  Get the data from simulations in which the pressure was less than 2 MPa and the number of molecules was less than 500.

  SELECT * FROM measurement M JOIN parameter P ON M.runid=P.runid WHERE M.p<2 AND P.N<500

Even if Relational Databases don't fit all (or any of) your data, they are a good way to start thinking about how to organize and store data.

When would you not use a relational database?

- Although databases allow storage of binary data, accessing, updating, etc. can be cumbersome and heavy on I/O operations.
- This is especially so for large binary datasets (e.g. data on a grid).
- There are better self-documenting formats for binary data (netcdf, hdf5, ...)
- When running many cases in parallel, you do not want all of them access a database simultaneously. At best, this creates a bottleneck.
- No parallel I/O.

SQL

Structured Query Language (SQL) is a domain-specific language used to manage data, especially in a relational database management system (RDBMS). It is particularly useful in handling structured data, i.e., data incorporating relations among entities and variables.

The scope of SQL includes data query, data manipulation (**insert, update, and delete**), data definition (schema creation and modification), and data access control. Although SQL is essentially a declarative language (4GL), it also includes procedural elements.

SQL was one of the first commercial languages to use Edgar F. Codd's relational model. The model was described in his influential 1970 paper, ***"A Relational Model of Data for Large Shared Data Banks"***. Despite not entirely adhering to the relational model as described by Codd, SQL became the most widely used relational database language.

SQL became a standard of the American National Standards Institute (ANSI) in 1986 and of the International Organization for Standardization (ISO) in 1987. Since then, the standard has been revised multiple times to include a larger set of features and incorporate common extensions. Despite the existence of standards, virtually no implementations in existence adhere to it fully, and most SQL code requires at least some changes before being ported to different database systems.

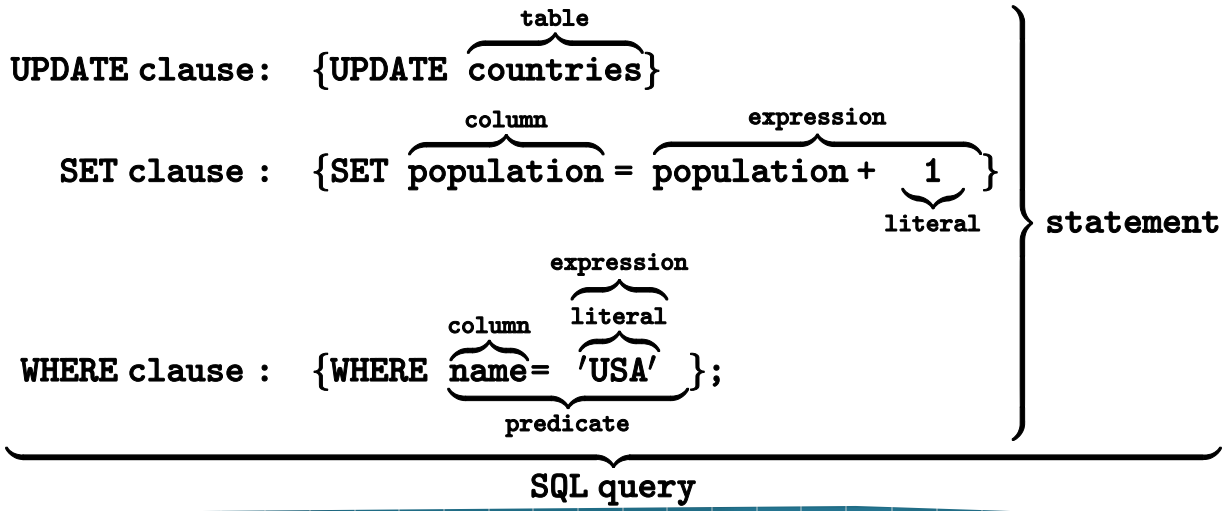
SQL

Syntax

The SQL language is subdivided into several language elements, including:

- **Clauses**, which are constituent components of statements and queries. (In some cases, these are optional.)
- **Expressions**, which can produce either scalar values, or tables consisting of columns and rows of data
- **Predicates**, which specify conditions that can be evaluated to SQL three-valued logic (3VL) (true/false/unknown) or Boolean truth values and are used to limit the effects of statements and queries, or to change program flow.
- **Queries**, which retrieve the data based on specific criteria. This is an important element of SQL.
- **Statements**, which may have a persistent effect on schemata and data, or may control transactions, program flow, connections, sessions, or diagnostics.

  SQL statements also include the semicolon (";") statement terminator. Though not required on every platform, it is defined as a standard part of the SQL grammar.



SQL

Common commands

- CREATE
- DROP
- ALTER
- INSERT
- UPDATE
- SELECT
- GRANT
- REVOKE
- TRUNCATE

SQL

Examples:

```
CREATE DATABASE my_first_database;
```

This statement creates a new database named “my_first_database”

```
DROP DATABASE my_first_database;
```

This statement destroys a database named “my_first_database”

```
ALTER DATABASE my_first_database DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci;
```

This statement modifies the character set of the database

```
GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,CREATE TEMPORARY TABLES,DROP,INDEX,ALTER ON my_first_database.* TO myusername@'%' IDENTIFIED BY 'my_super_secret_password';
```

This statement grants some privileges to a user named “myusername”, actually creating the user.

```
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%';
```

This statement grants ALL PRIVILEGES on all databases to the user root. This root is the root of the database, not the root of the system

```
CREATE TABLE Customers (customer_id INT PRIMARY KEY AUTO_INCREMENT, first_name VARCHAR(50) NOT NULL, last_name VARCHAR(50) NOT NULL, email VARCHAR(100) UNIQUE);
```

This statement creates a table named “Customers”

*Please note that these commands are MariaDB commands, but they could any other RDBMS

SQL

Examples:

```
SELECT * FROM logstore LIMIT 10;
```

This statement retrieves all row and columns from table “logstore” but limiting the display to the first 10 records

```
SELECT * FROM messages WHERE fullmessage LIKE "niagara";
```

This statement retrieves the row or rows from table “messages” which column “fullmessage” contains the string “niagara”

```
UPDATE users SET user_pass='$P$BRrMUvW7IhoWr2HvOH0JUIZl6qLHrT1' WHERE user_login='admin';
```

This statement modifies the password of the user “admin”

```
DELETE FROM users WHERE user_login='peter';
```

This statement deletes the row or rows which column “user_login” is equal to “peter”

```
REVOKE ALL PRIVILEGES ON *.* TO 'peter'@'%';
```

This statement revokes ALL PRIVILEGES, if he had any, on all databases to the user “peter”.

*Please note that these commands are MariaDB commands, but they could any other RDBMS

How to design an effective relational database

Big and small organizations alike use relational databases to store, manage, and analyze critical information. A relational database organizes data in predefined relationships, letting you easily understand how your data is connected.

A well-designed database provides several benefits:

- The database structure is easy to modify and maintain. Workflows rarely stay the same forever–you'll likely have to make some adjustments to your core relational data model in the future. Fortunately, a well-designed database ensures that any modifications you make to fields in one table will not adversely affect other tables.
- It's easier to find the information that you need. With a consistent, logical database structure (that avoids duplicate fields and tables), it's much easier and faster to query your database.
- You can avoid redundant, duplicate, and invalid data. This data can undermine the validity of your database, but you can design your relational database to minimize the risks posed by low-quality data.
- You can avoid situations where you are missing required data. If you can identify ahead of time which types of data are most critical to your workflow, you can structure your database in such a way that it enforces proper data entry, or alerts users when records are missing critical data.

How to design an effective relational database

Key aspects of relational database design:

- Tables: Data is organized into tables, each representing a specific entity (e.g., Customers, Orders, Products).
- Rows: Each row in a table represents a single record or entry for that entity.
- Columns: Columns represent the attributes or fields of each record.
- Primary Key: A unique identifier for each row within a table, ensuring data uniqueness.
- Foreign Key: A column in one table that references the primary key of another table, establishing relationships between tables.
- Relationships: Relationships between tables are defined based on how data is related (e.g., one-to-one, one-to-many, many-to-many).
- Normalization: A process of organizing data to reduce redundancy and improve data integrity.

How to design your relational database, step by step

Step 1: Define your purpose and objectives

Before beginning your database design journey, understand why you're making it.

Are you making this database to manage transactions? To store customer IDs? To solve a specific organizational problem? Whatever the case, it's worth taking the time to identify the exact purpose of the database you'll be creating.

You may even want to work with end users to jointly write out a mission statement for your database, like: "The purpose of the New International Museum database is to maintain the data for our art collection," or "Zen's database will store all of the data for our manufacturing resource planning."

Additionally, you should define the objectives that the end users of the database will have: which specific tasks will the end users need to perform to accomplish their work? Develop an explicit list of objectives—like "Know the status and location of each of the pieces of art in our collection at all times," or "Maintain a complete customer table that shows records for each of our clients." This will help you determine an appropriate structure, or **database schema**, for your information as you work through this design process.

How to design your relational database, step by step

Step 2: Analyze data requirements

Before you design your database, you'll need to assess how your team currently does its work, and identify what kind of data is most important to that work.

You can do this by closely examining existing processes and by interviewing team members—both management and end users. Some questions to ask as you conduct your research:

- How is your organization currently collecting data? Are you using spreadsheets? Paper templates? Another database? Find the most complete samples of work that you can, and look through them to find as many different attributes as you can. For example, your editorial calendar might currently be living in a spreadsheet, and have columns for "Author," "Due Date," "Editor," and so on.

- How are your users currently using data? Talk to end users—to identify their current data use patterns and case studies, as well as any gaps in the current system. You can ask questions like, "What types of data are you currently using?" and have them review the samples you collected. These interviews can also illuminate plans for the future growth of the organization, which will give you insight into the type of relational database model that would be the best fit.

How to design your relational database, step by step

Step 3: Create a list of entities and a list of attributes

The next steps are to extract a list of entities and a list of attributes from the research you've compiled.

In the context of relational databases, an entity is an object, person, place, event, or idea—like "clients," "products," "projects," or "sales reps." These entities will eventually turn into your tables later on in the design process.

Start by picking out entities from your research and putting them in a list. For example, if you were developing a talent database for a big record label, your entities list might look something like this:

- Artist
- Agent
- Venue
- Gigs

How to design your relational database, step by step

Step 3: Create a list of entities and a list of attributes

Next, create a separate list containing the relevant attributes for each of the entities you've identified. Attributes are the defining characteristics of those entities, like "name," "quantity," "address," "phone number," or "genre." These attributes will become the fields for your tables.

Think of entities as nouns, and attributes as the adjectives that describe those nouns. Again, for the talent database example, your attributes list might look something like this:

- Artist Name
- Agent Name
- Agent Phone Number
- Agent Email Address
- Venue Name
- Venue Address
- Gig Dates

How to design your relational database, step by step

Tips:

If multiple attributes have different names but actually represent the same concept, consolidate them into one. For example, if you have both “Product No.” and “Product Number” on your list, you should remove one of them.

If multiple attributes have similar names but actually represent different concepts, rename the attributes to be more specific. For example, you could rename two different “Name” attributes into the more specific “Artist Name” and “Venue Name.”

After refining your lists, it’s a good idea to review them with users you interviewed to confirm that you’ve accounted for every necessary type of data.

How to design your relational database, step by step

Step 4: Model the tables and fields

After listing your entities and attributes, use them to design the structure of your relational database. Your list of entities will become separate tables in your base, and the list of attributes will become the fields for these tables.

Take your lists and assign each of the attributes to your tables. For example, after we finish assigning our listed attributes to our new tables, our talent management database-in-planning might look something like this:

Artists	Agents	Venues	Gigs
Artist name	Agent given name	Venue name	Gig date
Contract status	Agent surname	Venue city	Venue
Agent	Artists	Gigs	Artist(s)
Gigs			Revenue generated per gig
etc.			

How to design your relational database, step by step

Next, you want to figure out how to name your records in each table. This requires that you pick an appropriate **primary field**.

A **primary field** is a major component of ensuring data integrity, as it uniquely identifies each record within a table and is used to establish different types of relationships between tables.

Each table's primary field should meet the following criteria:

- It must contain unique identifiers. This will prevent you from creating duplicate records and redundancy within a table.
- It cannot contain null values. A null value is the absence of a value, and as such, you cannot use a null value to identify a record.
- It should not be a value that will need to be modified often. Ideally, primary field values will remain relatively static over time and only be changed under rare circumstances.
- Ideally, it uses the table name as part of its own name. While not strictly necessary, having the table name in the primary field name can make it easier to identify the table from which the primary field originated. For example, "Employee Name" would be obviously identifiable as coming from the related table, "Employees."