Let us understand under normalization first. Assume that we are developing a simple Library Management System. We identify Book as the first entity, and start listing its different fields:

- `Book ID`
- `Title`
- `ISBN`
- `AuthorName`
- `PublisherName`

Now assume that we have created a Book table in the database with these fields, and entered a list of books:

BookID	Title	ISBN	AuthorName	AuthorAddress	PublisherName
1	Advanced C# 3.5	555-66	James Hetfield	33, 4th Street, CA	MTL Pvt Ltd
2	ASP.NET 3.5	4343-443	Kirk Hammet	56, Dersy Drive, NC	MTL Pvt Ltd
3	AJAX Simplified	334-34341	James Hetfield	33, 4th Street, CA	MTL Pvt Ltd
4	ADO.NET Internals	323-234	James Hetfield	33, 4th Street, CA	NPS Pvt Ltd

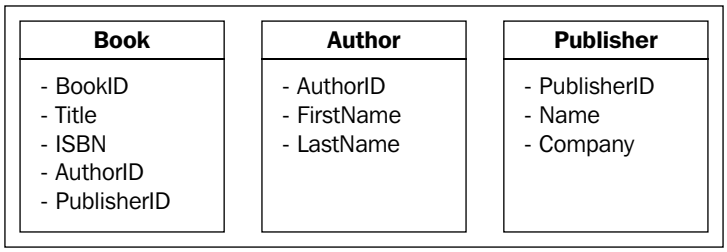
Now, there are some problems with this design, as listed below:

- If we want to have 100 rows of books written by the author James Hetfield, we will need to repeat `AuthorName` and `AuthorAddress` 100 times (once for each row), wasting valuable disk space and creating redundant data.
- If we want to retrieve a list of authors from the table above using an SQL query, we will need to make sure that we filter our repeated names.
- If we delete a book, the author would be deleted too, along with the publisher. Authors and Publishers are not dependent on the Books, so having the above design is a major drawback.
- If we need to update a publisher or an author, we have to update each row for every book published by that publisher or written by that author, which not only degrades performance but is also a cumbersome task.

To fix these issues, we need to normalize the data model and put the author and publisher details in separate individual tables. So instead of one Book table, we will have three tables:

- Book
- Author
- Publisher

This way, we will remove the dependency of Books on Authors and Publishers.



So now we have three normalized tables, and instead of repeating the authors in the Books table, we are now referencing authors from the Author table using `AuthorID` as a foreign key in the Book table.

When we normalize a table, we basically break it up into different tables. This means increased complexity when querying data as we now need to use multiple tables instead of a single table. We need to have a balance between normalization and performance of the queries (joins). For example, we know that many customers could have a common last name, but we cannot create another table for `LastName`. So sometimes we may want to 'denormalize' data for better performance.

Moreover, we don't need to normalize when archiving data. So data stored in archived tables (historical data) should be denormalized so that queries can run faster and be performance-efficient.

Data Modeling using MS Visio

Let us create a physical data model for our Order Management System from the logical data model we developed in this chapter. Although we will create a physical data model targeting Microsoft SQL Server 2005, you can use the same method to target any database you want.