DATABASE DESIGN

SUCCINCTLY

BY JOSEPH D. BOOTH

SUCCINCTLY EBOOK SERIES

Syncfusion®

Database Design Succinctly

By
Joseph D. Booth

Foreword by Daniel Jebaraj

Copyright © 2022 by Syncfusion, Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.
ISBN: 978-1-64200-223-2

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books 8

About the Author 10

Chapter 1 Introduction 11

Index cards to database tables 11

Spreadsheets 12

Small databases 13

SQL database 14

Summary 15

Chapter 2 Data Models 16

Conceptual data model 16

Logical data model 17

Entities 17

Attributes 17

Relationships 17

Physical data model 19

Summary 19

Chapter 3 Conceptual Model 20

Use cases 20

Implied entities 21

SIPOC diagram 22

Flow charts 22

Workflow diagrams 23

Whiteboard 24

Back of the napkin 24

Summary 25
Chapter 4 Logical Model 26
Attributes and keys 26
Type of keys 26
Data normalization 27
Flexible data 28
Redundant data 28
First normal form 28
Repeating groups 29
Create a new entity 29
Link to the primary table 29
Second normal form 30
Third normal form 31
Why bother? 32
Boyce-Codd normal form (BCNF or 3.5 NF) 32
Fourth normal form 34
Functionality dependency 34
Denormalization 35
Summary 36
Chapter 5 Physical Data Model 37
Tables 37
Naming 37
Fields 38
Data types 38
Constraints 42
Defaults 44

Computed fields....44
Logical entities to physical table....44
Customer table....45
Address table....48
Phone number table....50
Indexes....51
Views....51
Summary....52
Chapter 6 Data Standards....53
Internal standards....53
User defined type....53
User-defined table types....55
External standards....56
ISO standards....56
Address standards....59
Government standards....60
Summary....62
Chapter 7 Sample Data Models....63
Human resources....63
Conceptual model....63
Logical model....64
Accounting systems....65
Conceptual model....65
Logical model....66
Inventory....67
Conceptual models....67

Cost of goods sold 68
Logical model 69
Customer orders 70
Conceptual model 70
Logical model 71
Reservations 72
Conceptual model 72
Logical model 72
Sports 73
Conceptual model 74
Logical model 74
Summary 75
Chapter 8 Physical Model 76
Lookup tables 76
Journal type 76
Account type 77
Chart of accounts 78
Transaction 80
Entry 81
Putting it together 81
Summary 83
Chapter 9 Summary 84
Appendix A DBMS Field Types 85

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, CEO
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Joseph D. Booth has been programming since 1981 in a variety of languages, including BASIC, Clipper, FoxPro, Delphi, Classic ASP, Visual Basic, Visual C#, and the .NET Framework. He has also worked in various database platforms, including mySQL, PostgreSQL, Oracle, and SQL Server.

He is the author of 11 Syncfusion *Succinctly* titles including *Accounting Succinctly*, *Regular Expressions Succinctly*, and *SQL Server Metadata Succinctly*, as well as six books on Clipper and FoxPro programming, network programming, and client/server development with Delphi. His latest project is a SQL analysis tool called SQL Sleuth.

Joe has worked for a number of companies including Sperry Univac, MCI-WorldCom, Ronin, Harris Interactive, Thomas Jefferson University, People Metrics, and Investor Force. He is one of the primary authors of Results for Research (market research software), PEPSys (industrial distribution software), and a key contributor to AccuBuild (accounting software for the construction industry).

He has a background in accounting, having worked as a controller for several years in the industrial distribution field, but his real passion is computer programming. In his spare time, Joe is an avid tennis player, practices yoga and martial arts, and spends as much time as possible with his grandkids, Blaire and Kaia.

Chapter 1 Introduction

Data processing, computer information systems, and information technology. They're all phrases describing computer work, and they share a common theme—they work with data/information.

However, the way a user might perceive and use that information and the optimal way a computer system might store the information are often very different. In this book, we are going to discuss how to model the user's information into data in a computer database system in such a way as to allow the system to produce useful results for the end user.

Index cards to database tables

Sandy's antique store keeps its information stored in paper files or index cards. A computer would not fit the décor of her store. So, her store is a small business that relies on colored index cards: green for customers, yellow for orders, and red for products. Figure 1 shows the business filing system.

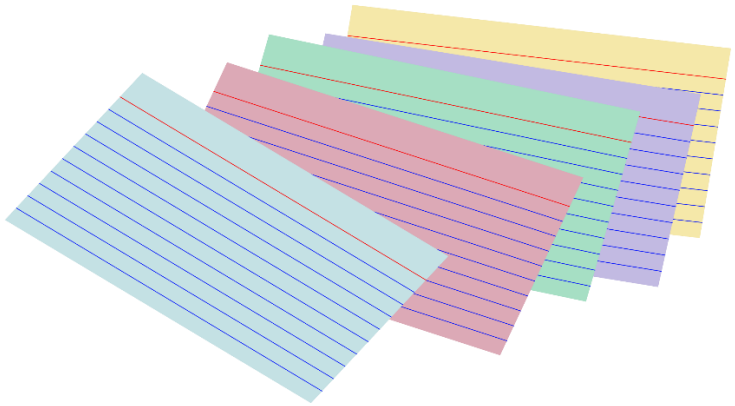


Figure 1: Manual filing system

The cards get updated for new customers and orders. This approach is very flexible for adding or updating information but makes it very difficult to retrieve information. When the business was told taxes needed to be retroactively collected for all Colorado orders, either sent to Colorado (yellow cards) or billed to Colorado (green cards), it was quite a task to determine the orders that should have taxes paid.

It was a slow process to go through all yellow (order) cards and put them aside if the address was in Colorado. For all the non-Colorado orders, she had to find the corresponding green (customer) card and check to see if the billing address was in Colorado.

Spreadsheets

Blaire's dental clinic decided that spreadsheets were a good solution, since one of their assistants knew Microsoft Excel well. Figure 2 shows part of the spreadsheet used to track patients.

	A	B	C	D	E	F	G	H	I
1	**Last Name**	**Father**	**Mother**	**Billed**	**Paid**	**Child 1**	**Child 2**	**Child 3**	**Child 4**
2	Smith	John	Kellie	$500.00	$250.00	Joe	Blaire	Kaia	
3	Alexander	Rich	Sandra	$800.00	$750.00	Ben	Matt	Mike	Susan
4	Keefrider	John	Kelly	$600.00	$600.00	Madison			
5	Fields	James	Jennifer	$1,200.00	$900.00	Mike	Sue		
6	...								

Figure 2: Dental tracking spreadsheet

Due to a recent grant, the clinic was told to give a $200 credit to any family with more than two children. This required a few changes and a formula to update the billed amount.

The following formulas were added.

=IF(COUNTA(F2:I2)>2,200,0) Determine families with more than two children.

=+D2-(E2+J2) Compute balance after credit.

The revised spreadsheet is shown in Figure 3.

	A	B	C	D	E	F	G	H	I	J	K
1	**Last Name**	**Father**	**Mother**	**Billed**	**Paid**	**Child 1**	**Child 2**	**Child 3**	**Child 4**	**Credit**	**New Balance**
2	Smith	John	Kellie	$500.00	$250.00	Joe	Blaire	Kaia		$200.00	$50.00
3	Alexander	Rich	Sandra	$800.00	$750.00	Ben	Matt	Mike	Susan	$200.00	-$150.00
4	Keefrider	John	Kelly	$600.00	$600.00	Madison				$0.00	$0.00
5	Fields	James	Jennifer	$1,200.00	$900.00	Mike	Sue			$0.00	$300.00
6	...										

Figure 3: Revised spreadsheet

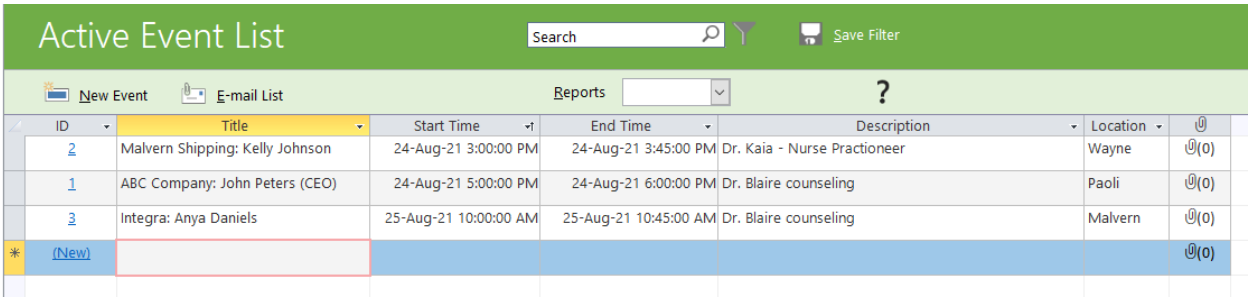
When the next patient came for a visit, with six children, the spreadsheet was revised even more. Soon, the clinic had to offer a refund to those families with a negative balance. The simple spreadsheet got more complex to handle the additional changes made at the clinic.

The use of formulas made it difficult for the average user to determine what the Credit column meant. So, in addition to the spreadsheet, sticky notes were added to the computer, telling people what some of the columns meant.

Note: In 1986, Lotus Development Corp. was sued because a construction company underbid a contract that they claimed was caused by improper formula handing in one of the Lotus products. The company dropped the lawsuit and Lotus claimed the deficiency was actually a feature.

Small databases

Chris's EAP (employee assistance program) company provides counseling services for employees of several different companies. They have chosen to track the appointments in a Microsoft Access database, shown in Figure 4.



ID	Title	Start Time	End Time	Description	Location	
2	Malvern Shipping: Kelly Johnson	24-Aug-21 3:00:00 PM	24-Aug-21 3:45:00 PM	Dr. Kaia - Nurse Practioneer	Wayne	(0)
1	ABC Company: John Peters (CEO)	24-Aug-21 5:00:00 PM	24-Aug-21 6:00:00 PM	Dr. Blaire counseling	Paoli	(0)
3	Integra: Anya Daniels	25-Aug-21 10:00:00 AM	25-Aug-21 10:45:00 AM	Dr. Blaire counseling	Malvern	(0)
(New)						(0)

Figure 4: Access event tracking

The system was easy for people to use for scheduling, and the counselor billing was handled by a third-party supplier. However, after a while, some client companies became concerned with privacy issues. They didn't want two people from the same company crossing paths on the way to and from counseling sessions.

At first glance, Chris felt it would be easy for the person doing the schedule. However, since the system did not force the schedule restrictions, it was up to the person doing the scheduling to get it right. And when the CEO of ABC company said "hello" to one of his employees on the way out, Chris realized he needed a better solution. He hired a developer to create a macro for his event table. Figure 5 shows the VBA editor and the start of the code that needed to be written.

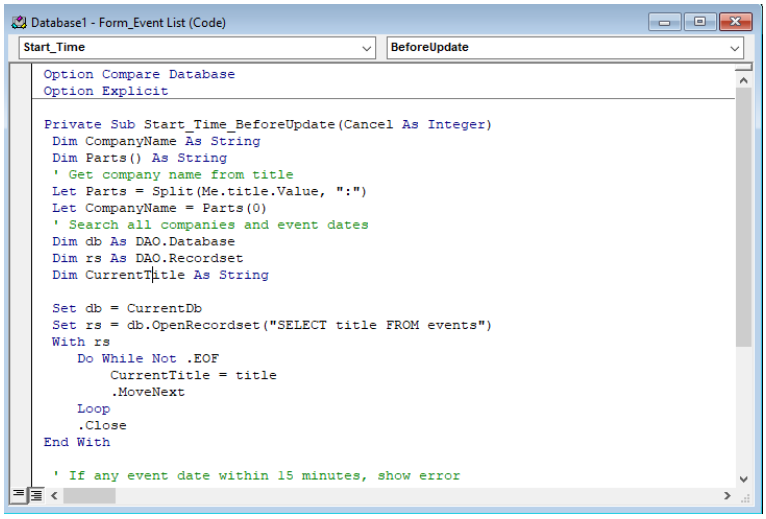


Figure 5: VBA Editor

Chris realized that as the company grew, more macros such as this were likely to be needed for his scheduling table. He debated whether he should bring a full-time developer on or rely on contractors to handle these ongoing fixes to his Access database.

SQL database

Christy's tennis club was doing well, and as it grew, she decided to store her records in a SQL database system. Her brother-in-law, John, took some database courses, so she asked him to create the database. He added a table to track members, as shown in Figure 6.

MemberId	FirstName	LastName	SpouseName	Children
1	John	Rider	Kelly	Susan,Jim
2	Joe	Scot	NULL	NULL
3	Rich	Alexander	Sandra	Denise,Rich,Pete
4	John	Rider	Kelly	Susan,Jim
5	Tom	Harris	Susan	Jim, Tom, Bill, Michelle
6	Rachel	LeForge	Mike	Alexis; Daniel

Figure 6: Tennis club members

A bit later, Christy wanted to determine how much money she could potentially make with a one-time offer of a $20 membership fee for each family member. Since the club had over 1,000 members, she asked John to find out the total number of family members (kids and spouses) within the club. John eagerly took to the task, but soon found it was very difficult to count the family members. He came up with the following code (partly borrowed from the internet).

```
-- Count spouses
select 'Spouses' as Whom, count(*) as SpouseCount
from dbo.Members
union
-- Count children
select 'Children',
sum(len(children)-len(replace(children,',',''))+1)
from dbo.Members
```

He happily showed the results (Figure 7) to Christy.

Whom	SpouseCount
Spouses	6
Children	12

Figure 7: Number of family members

Christy took the numbers and made her offer to club members. Unfortunately, the numbers from his query are wrong.

Tip: As an exercise, see if you can identify the problem with the query John wrote.

Summary

The examples here show some information systems that, while flexible, begin to run into real difficulties for reporting and custom programming. While index cards can always hold more information, Excel columns can be added, and Access macros can be written, designing the database properly with a good database management system can offer much more robust reporting and control.

Even if the data is stored in a database system, it still can be difficult to work with and possibly inaccurate when the design is wrong.

In the next few chapters, we will cover how to design a database system to allow businesses to get better reporting and control over their information. We will also address how to improve the data that is put into the system, to make sure it is as accurate as possible.

Chapter 2 Data Models

To understand how to make a user's view of their information and create a database system, we rely on three primary models: conceptual, logical, and physical. Each model has a different audience and role. In this chapter, we will briefly describe the models, who uses each one, and the role that typically creates each model.

Conceptual data model

A conceptual data model is a very high-level view of the entities and actions that the database system will model. It is intended to be very nontechnical and not very detailed. It should provide just enough information for a person to get an overview of the type of data and processes the system will contain.

A conceptual model should be created very early in the data modeling process. It generally relies on the defined use cases to extract the entities that the system will model. A business analyst or manager typically is the primary resource for defining the conceptual model. Figure 8 shows a simple conceptual model for an ordering system.

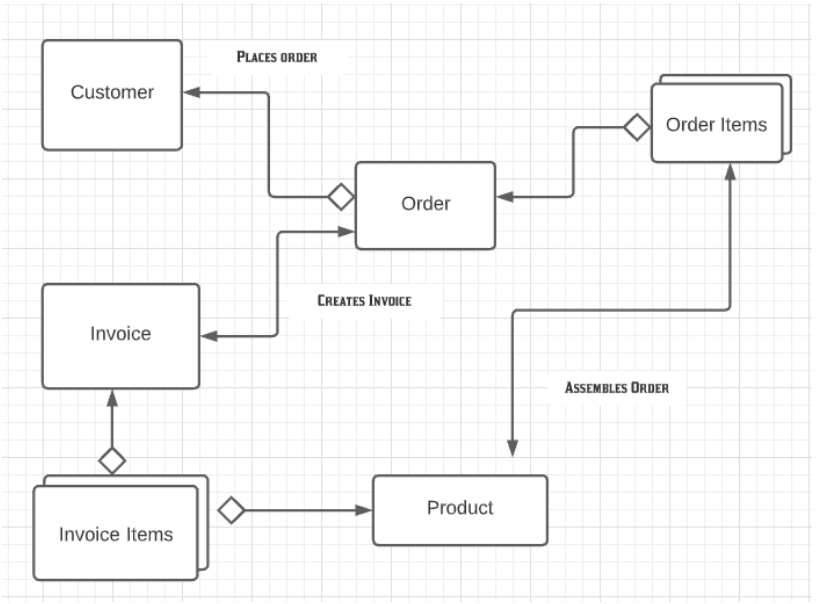


Figure 8: Conceptual model (order system)

Very little detail is provided by this model; it's just an overview of the primary entities (customers, orders, and invoices) and relationships (customer has multiple orders, the order has one invoice) among those entities.

Logical data model

The logical model consists of entities, attributes, and relationships between entities. It does not address the physical implementation, but rather should be database-agnostic. A good logical model could be implemented in any database management system. You should also be able to design the model using an object database or even a collection of flat files.

Entities

An entity is a person, place, or thing of which the model needs to keep track. Each entity will typically be represented by one or more tables in the physical model. For the logical model, we want each entity to have a name and the attributes of that entity. In addition to attributes, one or more attributes will serve as a key (the unique identifier for the entity).

Attributes

Attributes are properties associated with an entity. For example, a customer entity most likely has a name, address, and phone number. In the logical model, we are defining attributes at a higher level than we need for implementation. The following table shows some examples of logical versus physical attributes.

Table 1: Logical versus physical attributes

Logical attribute	Physical model
Address	Name, Address Line(s), City, State, Country, Postal Code
Name	First, Last, Middle Initial, Salutation
Business unit	Organization, Division, Department, Website URL

Ideally, the logical attributes will have an associated standard definition of the components of the attribute (so all address attributes, regardless of entity, should look the same).

Relationships

Relationships in the logical model describe how entities interact with each other. There are three relationship types to consider.

One-to-one (1:1)

There is a direct relationship between the two entities and a single connection. For example, an order is associated with a single customer. An invoice is associated with a single order, and so on. Note that a 1:1 relationship only refers to a single occurrence. A customer may have multiple orders, but each order is associated with only one customer.

One-to-many (1:n)

A one-to-many relationship occurs when an entity can have multiple occurrences of another entity. In our example, a customer has a one-to-many relationship with orders (so a customer can have many orders), but each order only has one customer. Similarly, the order has multiple line items, but each line item is linked to a single order.

Many-to-many (n:n)

Our product entity might be supplied by multiple vendors, and each vendor might supply multiple products. This is a many-to-many relationship between the two entities. Most relational databases don't directly support this type, but it is a fairly common practice to define this type using a third table, sometimes called a junction table, to link them together.

Figure 9 shows our logical model for the ordering system based on the conceptual model we defined earlier.

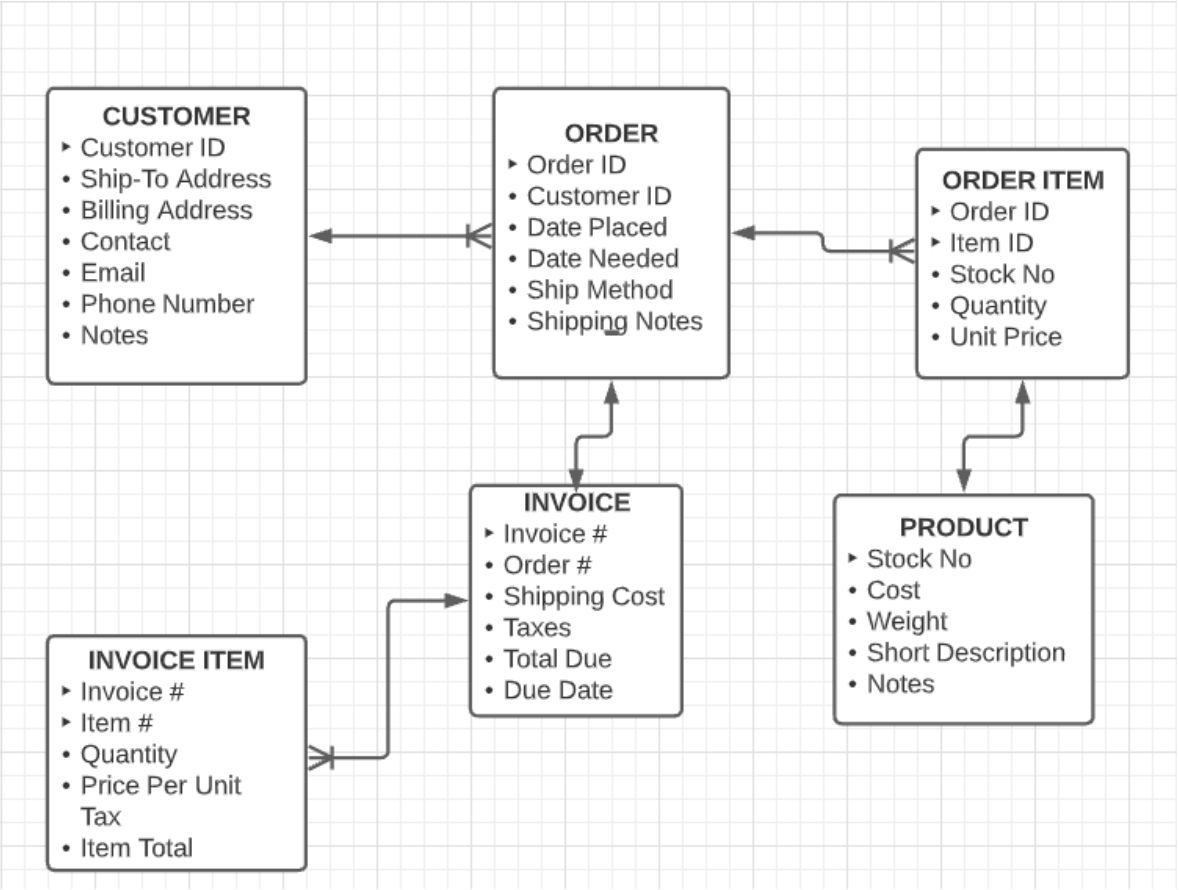


Figure 9: Logical data model

Physical data model

Once the logical model is completed, a database developer or administrator will implement the logical model as tables and columns in a database management system. Most entities will be represented by a table (or multiple tables if needed), and the relationships will be defined via indexes and foreign keys.

At this point, we are concerned with the target database, and the developer will likely create the model using the columns and constraints that the target database uses. While the SQL language is common among SQL dialects, there are variations among the various products.

Figure 10 shows our physical model for the ordering system, derived from the logical model we designed earlier. The model assumes Microsoft SQL Server is the target database.

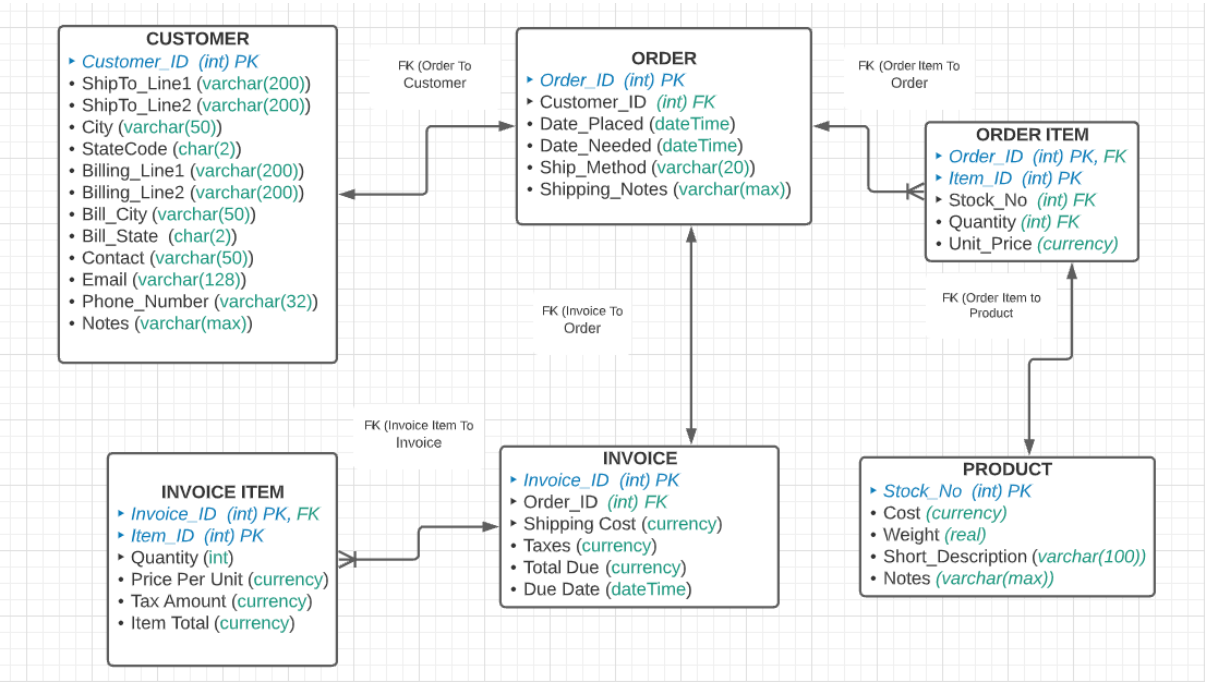


Figure 10: Physical data model

The physical model might contain additional information, such as database constraints, nullable columns, and views versus tables. We will cover this in more detail in a later chapter.

Summary

The three models described here take the customer's understanding of their information and translate that understanding into a model that can be implemented in a database.

Note: We are focusing on relational databases, such as Microsoft SQL Server and Oracle. Nonrelational databases, such as MongoDB and Redis, offer a different approach than tables to hold the data.

Chapter 3 Conceptual Model

In this chapter, we will explore how to create a conceptual model.This is the start of the requirements-gathering phase. Although gathering might be too gentle of a description; it is more like hunting the requirements down. Often, you will need to work with multiple people to get an overview of the business, which is how the conceptual model will get created.

As you work through the business process documentation, you should keep an ongoing list of entities (people and things) and relationships among the entities (usually derived by understanding the actions that occur among the entities). I generally include these items when reviewing the business processes:

- Entity: Person or thing to be modeled.
- Relationship type: Relationship to another entity (one-to-one, one-to-many, many-to-many).
- Related entity: Other entity related to the first entity.
- Action: Short, descriptive text of relationship.
- Source: Person or department that provided the information.
- System: System (HR, order, finance, and so on). Entities are part of this higher system.

One key idea is to keep your conceptual model simple, but thorough. The goal is to produce a quick, easy-to-use overview. It is better to have several smaller conceptual models than one large model that quickly becomes difficult to follow.

It is also a good idea to know where the information came from. The warehouse model starts with the order, and while at the warehouse they know the order is associated with a customer, they most likely are not familiar with any credit checking that gets done before the order is placed. Just as likely, the finance department generally does not know the details of the warehouse and shipping department.

Use cases

A use case is a written description of actors and actions they perform. It often includes a name, description, and diagram. It includes the actor (role acting), any necessary preconditions, and the workflow. For example, a customer placing an order might look like Figure 11.

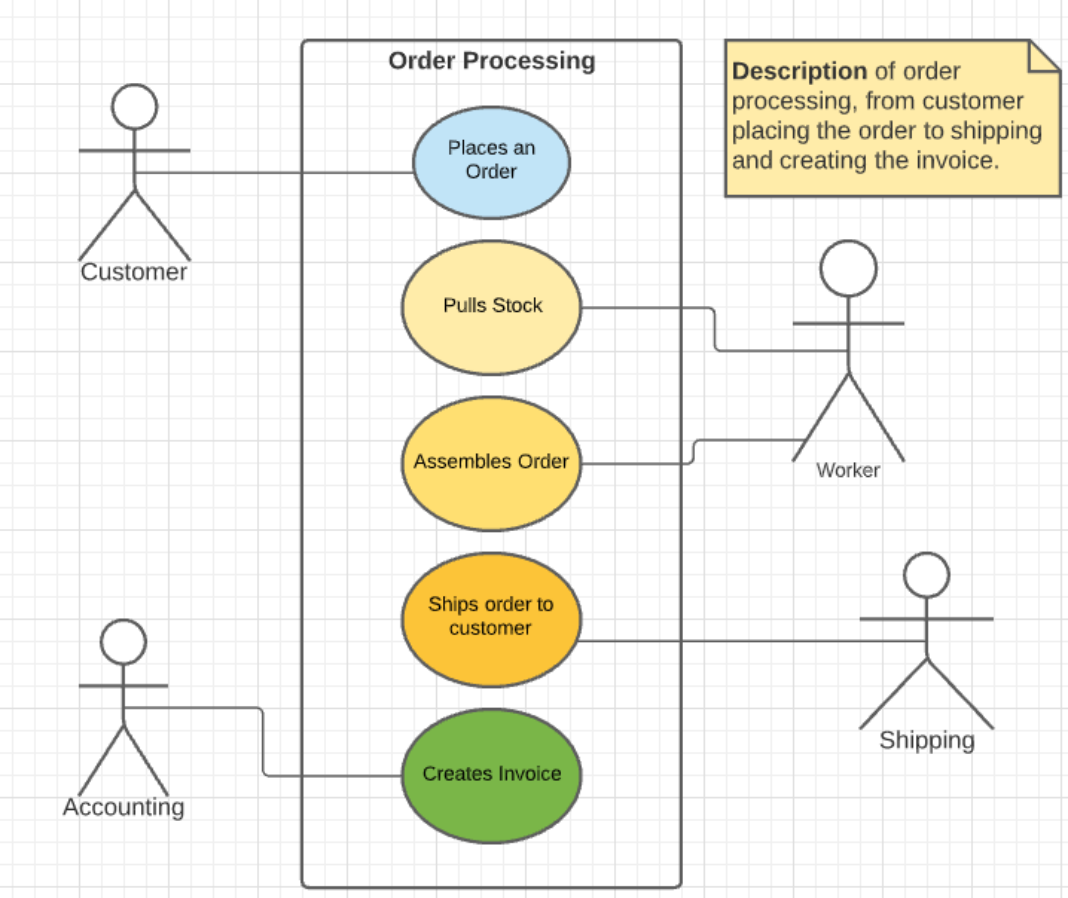


Figure 11: Order use case

From this diagram, we can extract entities (people and things). We see a customer, some employees, an order, stock, and invoice entity. Not all entities need to be modeled; for example, we might not need employees, but rather departments.

We can determine a relationship between the customer and the order, the order and both the warehouse (worker) and shipping department (shipping). There is also a relationship between the invoice and the accounting department.

Note that what doesn't appear in the diagram, but might be implied, is a relationship between the invoice and the customer (or the invoice and the order itself, depending upon whether an invoice can cover multiple orders).

Implied entities

While the use case appears complete, you should ask questions. For example, does the shipment always go to the same location? If the answer is no, then there is an implied shipping address associated with the customer with a one-to-many relationship (one customer to many shipping addresses) between the two entities.

SIPOC diagram

Another tool to consider is the SIPOC diagram. This is a Six Sigma tool used to document business processes. The acronym stands for suppliers, inputs, process, outputs, and customers.

The Suppliers column shows the people, departments, etc., that create an input. For an ordering system, the suppliers are the customer, who creates the order input; the warehouse workers, who will pick and assemble the order; and finally, the finance department, who will create the invoice.

Table 2: A sample SIPOC diagram for ordering

Suppliers	Inputs	Process	Outputs	Customers
Client	Order	Calls or places order via the website.	Items ordered	Warehouse
Warehouse	Items ordered	Gather items ordered. Place in shipping boxes. Add shipping label.	Assembled shipment	Shipping
Shipping	Shipment	Create invoice for items. Mail invoice to the customer.	Invoice	Finance

From this chart, we see the client or customer entity, the order entity, and the invoice entity. We can also see a list of items on the order that needs to be generated, both for the shipping and the finance departments.

We can also see relationships, such as the client being associated with the order, and the order line items being linked to the order entity. The invoice is tied to the order and a customer.

Flow charts

A flow chart is a series of symbols and connecting lines that describes a process. The processes are defined as rectangles, and the decision is a diamond with Yes and No pathways. The user can follow the flowchart steps to determine the process. In Figure 12, we have an example of the process flow to determine if a customer has a good credit rating before placing an order.

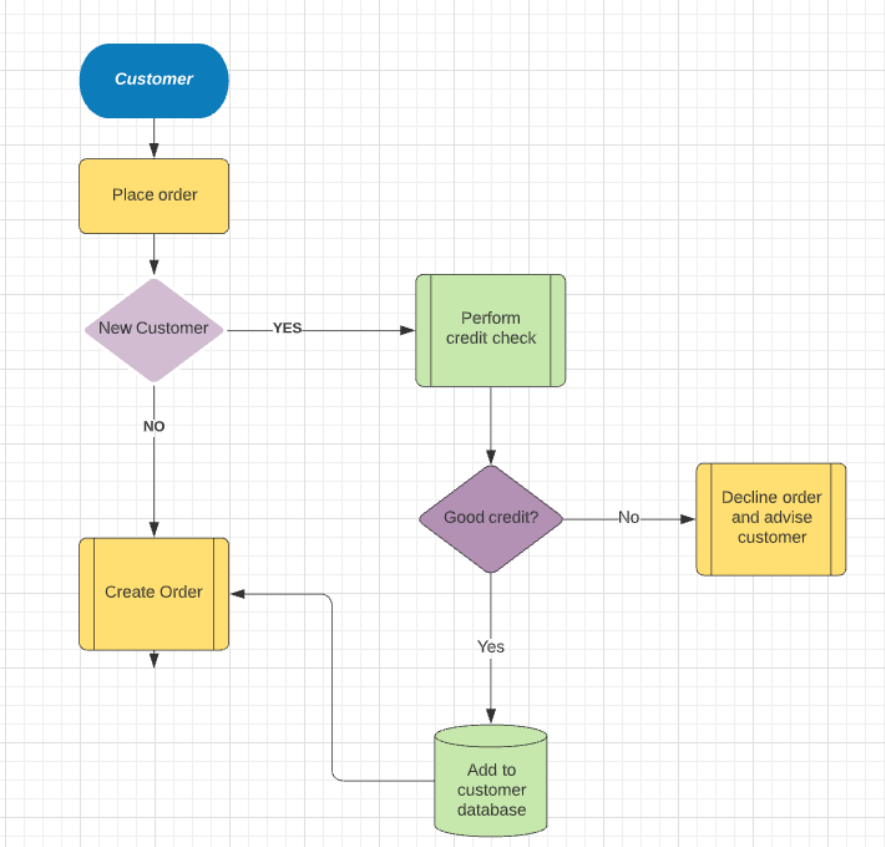


Figure 12: Credit check flowchart

The customer and order entities are defined on the flowchart, and the finance or accounting department is implied (someone needs to perform the credit check). We can also see that the customers are saved in a database. This is done from the accounting department.

Workflow diagrams

A workflow diagram is a graphic image depicting the departments, artifacts, and steps that define a business process. They show a linear flow, with some Yes/No or True/False branching as needed. Microsoft Visio allows users to define their business processes as workflow diagrams.

Figure 13 shows a sample of a workflow diagram, showing a customer placing an order via a website, and the order being sent to the finance department for approval (courtesy of Erica Quigley from her business process book).

Figure 13: Workflow diagram

From the workflow diagram, we have identified the customer and finance department entities, and the website entity is where the order entity is defined.

Whiteboard

Sometimes you might need to have a meeting with various departments and sketch out the steps in the process on a whiteboard. Often in these types of meetings, a lot of the “steps” the departments just do, get spelled out. For example, when describing the process of assembling an order, a good facilitator might drill down and discuss what to do when an item being requested is backordered. If it happens infrequently, the employees might not even think about it, but it is still an important piece of information to determine.

Figure 14: A whiteboard session

Back of the napkin

Let’s say you encounter a person who has been doing their work a long time, but they’ve never written the actual process down. In this case, you will likely sketch the process learned from that conversation onto a napkin, rather than formal documentation for the process.

Summary

You will likely encounter a variety of different ways that business processes are documented, and just as likely, find processes without documentation. The gathering-of-requirements phase will likely be a mishmash of documents, diagrams, and conversations. As you understand the businesses processes, the components of the conceptual model should become clear.

It is important not to skip or gloss over the conceptual model. Some business processes occur infrequently and can be easily missed if you jump directly to the logical model. Be sure to understand all the processes you need to create the database for, not just the most common processes.

Once you've determined all the entities and relationships from your gatherings, you can build your conceptual model. Place the entities on a diagram, with simply an entity name. Draw lines between the entities, thinking in terms of one-to-one or one-to-many links between them. Typically, you will label the line with a brief word or two explaining how they are linked.

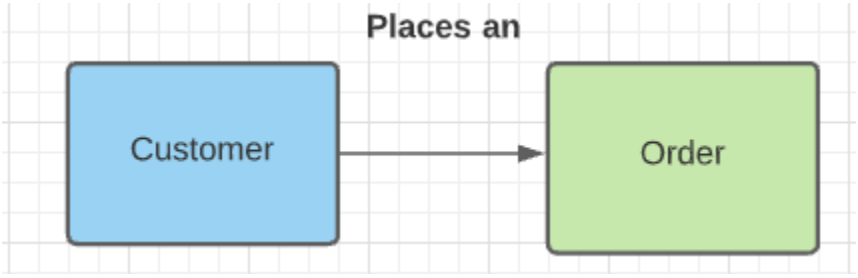


Figure 15: Two entities linked together

Do not create a large conceptual model; it is difficult to understand. It is better to have two separate models when the systems are not related, than one monolithic model that will be very difficult to grasp quickly.

Note: If you'd like to learn more about business process modeling, check out Erica Quigley's book,* Business Flow Process Mapping Succinctly*, on the Syncfusion site.

Chapter 4 Logical Model

The logical model begins to flesh out the details from the conceptual model. Attributes are added to the entities, and relationships are more clearly defined. We still aren't concerned with database implementation, but rather providing the details of what information the database must model.

In this chapter, we will take all the entities gathered previously and define a logical diagram to be the basis for the actual database design. However, we want to make sure that our model is both flexible and robust, to make sure the database can be used efficiently by the business.

Attributes and keys

When describing an entity, there are two interrelated items to consider. Attributes are the elements of the information associated with the entity. Each attribute should contain a single fact about the entity. For a customer, this might include name, address, and tax information. For an order, the attributes might be the order number, date placed, or date required.

The keys are either a single attribute or a combination of attributes that allow an entity to be uniquely defined. You could create a key of every attribute, since that should be enough to identify the record. So technically, every attribute is also a key. However, the best key is the smallest attribute or collection of attributes necessary to identify the entity uniquely.

Type of keys

To better understand keys, let's consider the various keys that you'll need to work with.

Primary key

Every entity requires a primary key. This is the selected candidate key that will be used. It must be unique and cannot be NULL.

Candidate keys

Candidate keys are attributes that are minimally necessary to identify a record. For example, if we are modeling an employee with the following attributes:

- Employee ID
- Social Security number
- First and last name

Each of these could be considered a candidate key. The attributes hold just enough information to identify the entity, without any extra information. By identifying the candidate keys, you can select one of them to become the entity's primary key.

Superkey

A *superkey* is a collection of attributes that uniquely identify the entity but may contain extra, unneeded information. For example, in our person entity, the first name, last name, and age together form a superkey, even though the age attribute doesn't help identify the entity. An entity will likely have many superkeys.

Composite key

If more than one attribute is needed to create the key, then that is a composite key. If the first and last names were separate fields, and you want to use that as a key, it would be a composite key.

Surrogate key

For various reasons, you might not want to use any of the candidate keys. In those cases, you can create an internally generated key that has no actual meaning, but simply is the primary key to identify the record.

For example, if we chose first and last name as our primary key, other referencing entities would need to store those values to find the related entity. If the person's last name changes for whatever reason, every entity would need to be updated. In such an example, a simple integer value that would never change could offer a better solution to joining the entities together.

Foreign key

A foreign key is a linking key used to create a relationship between a child record and its parent entity. An order entity must have a customer, so the primary key of the customer entity will be stored as an attribute (foreign key) in the order entity. A foreign key must reference a primary key in another table or be NULL.

Note: Most relational database systems allow you to define what happens if the parent record is deleted. You might not allow the deletion if the parent has child rows, or you might automatically delete the child rows, and so on.

Data normalization

One of the first concepts in the logical model is a process called *normalization*. This is a step-by-step process designed to make your database more flexible and robust, and to eliminate redundant data.

Normalization is a set of rules (called *normal forms*). Once a rule is applied, the database is in "n" normal form. Most databases will be designed to third normal form, even though additional forms (rules) exist. We will briefly describe the additional forms in this chapter, but for practical purposes, the third normal form should be good enough.

Flexible data

The database design should not impose arbitrary limits on the data. For example, when tracking the number of children a person has, the system should allow no children up to some practical number. A business rule might set the maximum number of children, but the database design itself should not create a limit.

Redundant data

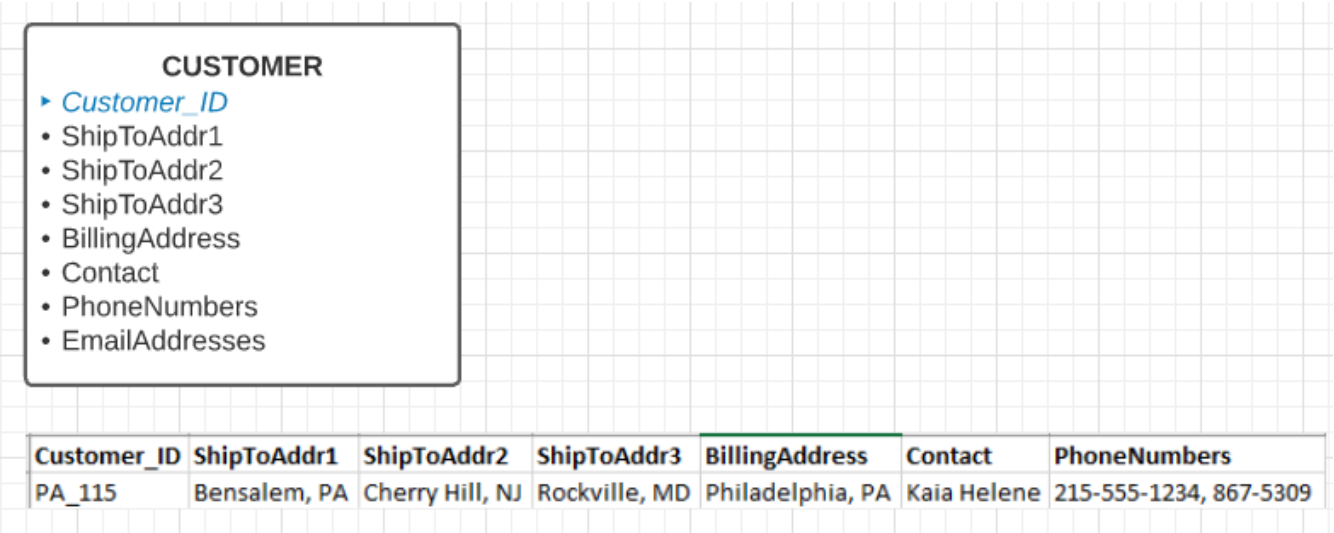
Redundant data (values that appear in multiple tables and fields) not only wastes space, but also makes maintenance in the database more difficult. Imagine in the real world that the finance office keeps track of customer information (including ship-to addresses). The warehouse also tracks customer information (to know where to ship products). However, the people in the warehouse often can't reach finance, which is why they keep the addresses themselves.

When the customer informs the company (via the finance office) of the new location, the warehouse might not be informed and could ship the product to the old location. In this type of scenario, the address is kept in two locations, and both must be updated when a change is made. Normalization will remove the redundant information, keeping the ship-to address in one spot.

First normal form

The first normal form (rule) primarily deals with flexibility in the database. The rule has three steps that should be completed:

1. Identify any repeating groups.
2. Create a separate table for the items in the repeating group.
3. Link the separate table to the parent record.



Customer_ID	ShipToAddr1	ShipToAddr2	ShipToAddr3	BillingAddress	Contact	PhoneNumbers
PA_115	Bensalem, PA	Cherry Hill, NJ	Rockville, MD	Philadelphia, PA	Kaia Helene	215-555-1234, 867-5309

Figure 16: The original table for the customer

Repeating groups

Any repeating groups in a table must be removed. For example, our customer table might have the following attributes:

- Customer Name
- Billing Address
- Ship-to Address 1
- Ship-to Address 2
- Ship-to Address 3

If every customer had exactly three shipping addresses, this might be feasible (despite other problems it causes), but that is very unlikely to be the case. Some customers might only order electronic products and/or services, so, therefore, they don't need a ship-to address. Other customers might have a shipping location in 20 different states or provinces, but this design prevents the customer from having more than three locations.

Repeating groups might also be stored in one column that has multiple values. For example, you might have a field called Customer Phone that has multiple phone numbers separated by commas. Table 3 shows an example.

Table 3: Multiple values in a single field

Customer Name	Billing Address	Customer Phone Number
ABC Company	Malvern, PA	610-555-1234, 215-867-5309

This violates the concept that each attribute should contain a single fact about the entity. In either design, we need to fix the model to eliminate the repeating groups.

Create a new entity

Once you've identified the repeating groups, create a separate entity for the items. For our shipping address, our new entity might look like this.

Customer ID (Foreign key to customer table primary key)
Ship-to Name (or number) (This, with the customer ID, is our composite, primary key)
Address information

For our phone numbers, we might have a customer ID and phone type (mobile, work, home, fax, and so on). Customer ID and phone type would be the composite primary key for the customer phone numbers entity.

Link to the primary table

The customer ID is the primary key attribute of the customer entity. The ship-to name (which could be a type or number) is combined with the customer ID to create a composite primary key for the shipping entity.

With this step done for all repeating groups, the table is in first normal form. You can have any number of shipping addresses, phone numbers, and so on. Customers with no shipping location can be represented, as well. Our new logical model for a customer entity is shown in Figure 17.

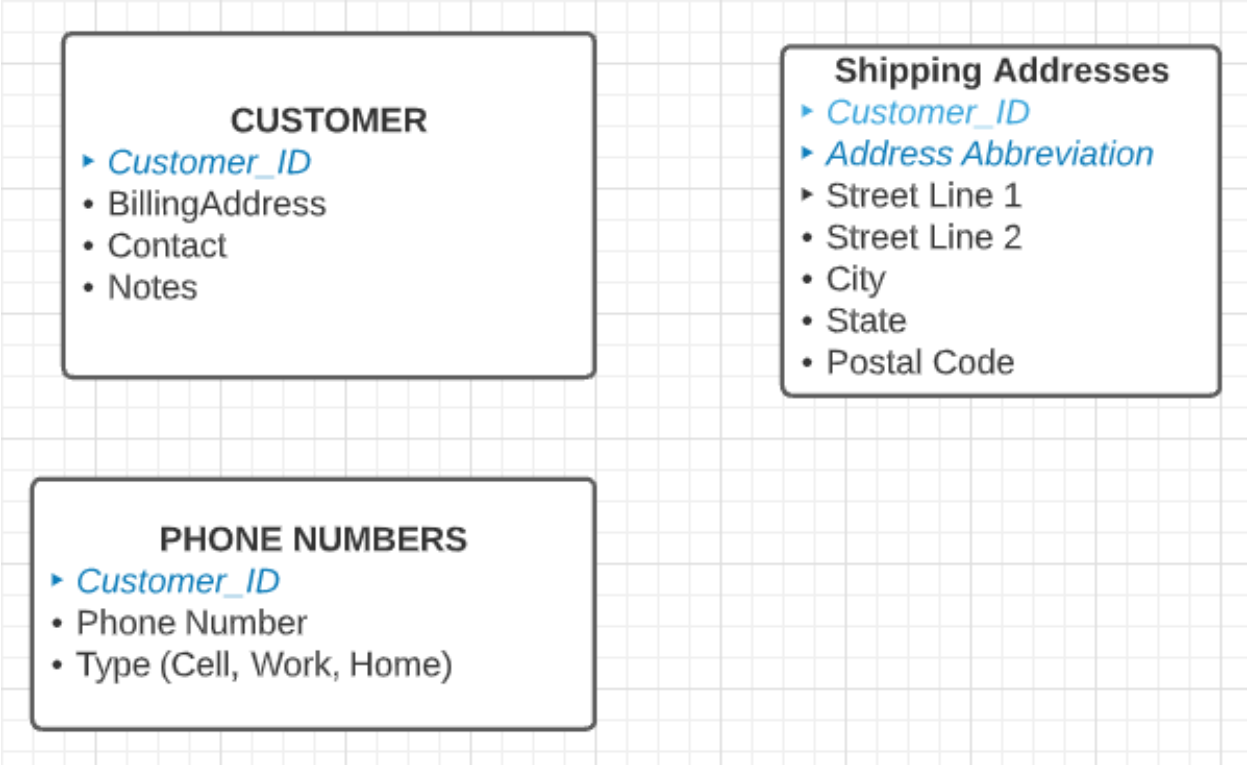


Figure 17: Revised logical customer

The phone number table implies that we only have one phone number of each type. For practical purposes, we might create an integer value for each separate phone number (so the composite key would be the customer ID and integer value). The phone type would become an attribute of the entity, not part of the key.

Second normal form

The second normal form addresses some potential redundancy in the database design. For a database to be in second normal form (2NF), it must already be in first normal form (1NF). After that, we need to apply the following rule.

Split partial dependencies to separate tables

A partial dependency occurs when an attribute (such as Ship-To) depends on part of the key, not the entire key. For example, consider our order line-item table. Each order consists of one or more items, so the initial entity might look like Table 4.

Table 4: Partial dependencies

Order Number	Seq	Qty	Ship-To	Product Code
100	1	4	Bensalem, PA	TB
100	2	2	Bensalem, PA	WRIST
214	1	12	Cherry Hill, NJ	TB

In this example, the ship-to location is only based on the order number, not the combination of Order Number and Seq (primary, composite key). So, the Ship-To field is redundant and should not be part of the line-item entity.

We can solve this by creating a new entity if needed. Or we might be able to determine the Ship-To location directly from the order. In this example, the Ship-To information is stored with the order, so a new entity is not needed.

Note: If the system allows line items to be shipped to different addresses, then the Ship-To column is not redundant. It is not redundant if both parts of the key are needed to determine its value.

Third normal form

The third normal form (3NF) addresses another potential redundancy in the database design. For a database to be in the third normal form, it must already be in the second normal form (2NF). After that, we need to apply the following rule.

Remove any attribute columns not dependent upon the primary key.

For example, Table 5 shows a potential design for the order line-item table. The primary key for this table is the combination of the Order Number and the Seq number. The quantity and the product code both are dependent upon the combined key.

Table 5: Order line-item table

Order Number	Seq	Qty	Product Code	Product Name	Cost
100	1	4	TB	Tennis Balls	$2.99
100	2	2	WRIST	Wrist Bands	$5.00
214	1	12	TB	Tennis Balls	$2.99

However, the Product Name and Cost are only based on the Product Code, not the combination of the Order Number and Product Code. Both Product Name and Cost are redundant.

We can solve this by creating a new entity, Product, that has the Product Code as the identifying key, and Product Name and Cost as attributes.

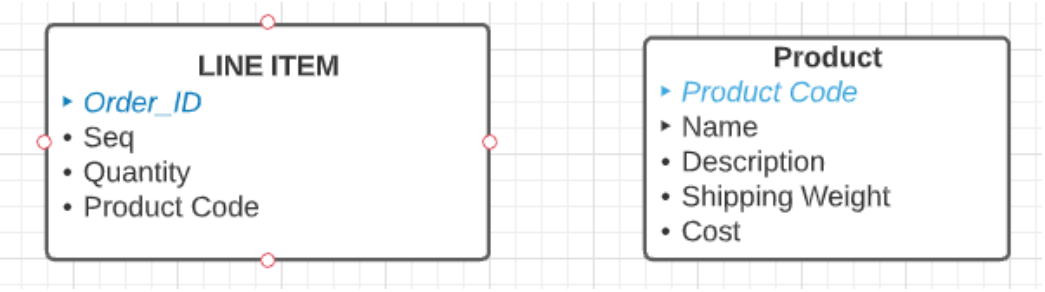


Figure 18: Revised logical design

In this example, we added the shipping weight one time to the product table, as opposed to needing to modify the line-item table and update the shipping weight for all ordered products.

Note: Working with prices can be complicated, as prices can change, and we need to ensure line items for prior orders show an accurate price at the time the order was placed. We can do this by either keeping multiple price points in the product table (and having a composite key of the code and price point) or keeping multiple price points with a change date in a separate table. To show the line-item price, we need to check the order date to see what the price was before any later change.

Once a database design is in third normal form (3NF), most times this is sufficient to start the database implementation.

Why bother?

What have we solved with these steps? By moving the database to first normal form (1NF), we have addressed two issues. Each attribute should be a single fact about the entity, and the database is not restricting the number of attributes that can be associated with the entity.

When the design reaches the third normal form (3NF), updates to attributes only need to be done in one spot, which reduces the possibility of different values for the same conceptual attribute. If we kept the product name and cost in the line-item table, then updated the cost on the first line item, but not the second, asking the system the cost of tennis balls would result in two different values.

Boyce-Codd normal form (BCNF or 3.5 NF)

This form is an improvement over 3NF if your table has multiple attributes (columns) in the primary key. If the primary key has only a single attribute, the entity is already in BCNF.

The attributes of the table are considered prime attributes if they are part of the key, and nonprime if they are not. If a nonprime attribute is dependent upon a single prime attribute (not the entire key), this violates BCNF and should be corrected.

For example, imagine a tennis club has members who take classes and clinics run by the club pros. Our table might look like Table 6.

Table 6: Tennis lessons

Member	Class	Pro Name
Christy	Women's 4.0 Mixed	Luis
Christy	Sunrise Tennis	Vince
Rachel	Women's 4.0 Mixed	Chris
Sondra	Competitive 4.0 Drill	Mike
Mary	Sunrise Tennis	Vince

The primary key for this table is the Member and Class attributes, so these are our prime attributes. The Pro Name is a nonprime attribute. In this example, the Pro Name is dependent upon the class, but not the full key. BCNF is violated when a nonprime attribute (Pro Name) is dependent upon a prime attribute (class), but not the entire key. To put this into BCNF, we create a separate table of **Pro Name** and **Class**, as shown in Table 7.

Table 7: Classes

ID	Class	Pro Name
1	Women's 4.0 Mixed	Luis
2	Sunrise tennis	Vince
3	Women's 4.0 Mixed	Chris
4	Competitive 4.0 Drill	Mike

Finally, our table for lessons will look like Table 8.

Table 8: Tennis lessons

Member	Class
Christy	1
Christy	2
Rachel	3
Sondra	4

Member	Class
Mary	2

Fourth normal form

The fourth normal form is used to address potential redundancy that could occur with multivalue dependencies. Since this situation is considered to rarely occur, most normalization effort stops before 4NF.

Functionality dependency

The first 3.5 normal forms deal with functional dependencies. A functional dependency is when the attribute is dependent upon the entity key. For our customer table, we can say that the name, address, city, and so on are attributes that are functionally dependent upon the customer ID key. Given a customer ID, we can determine the other attributes.

A case called a multivalue dependency occurs when two or more attributes together are associated with the entity key. It is the combination of attributes that is associated with the key, and not either attribute by itself.

As an example, a college provides a list of courses. Each course has a teacher and a book associated with it. There is no relationship between the teacher and book directly; they are only related to the associated course. Table 9 provides an example course entity, showing the teacher and required book for each course.

Table 9: Course entity

Course	Teacher	Book
Machine Learning	James McCaffrey	*Python Machine Learning*
Machine Learning	Ed Frietas	*Machine Learning using C# Succinctly*
Business Processing	Erica Quigley	*Death to the Org Chart*
Business Processing	Blaire Glacken	*Business Process Flow Mapping Succinctly*

If we wanted to see the teacher and book for a given course, the result would imply a relationship between teacher and book, where one does not exist. The relationship only exists in the course—there is no direct correlation between teacher and book.

To solve this implied relationship issue, we need to put these tables into the fourth normal form (4NF). This requires breaking the multivalue attributes into separate entities to prevent the implied relationship that does not exist. Tables 10 and 11 show the course entity placed into the fourth normal form.

Table 10: Course/Teacher entity

Course	Teacher
Machine Learning	James McCaffrey
Machine Learning	Ed Frietas
Business Processing	Erica Quigley
Business Processing	Blaire Glacken

Table 11: Course/Book entity

Course	Book
Machine Learning	*Python Machine Learning*
Machine Learning	*Machine Learning using C# Succinctly*
Business Processing	*Death to the Org Chart*
Business Processing	*Business Process Flow Mapping Succinctly*

While this design now requires two queries to get the teacher and book associated with the course, it prevents the false implied relationship between the teacher and the book.

Denormalization

Although the normalization process addresses redundancy in your database design, the extra tables that might be required could impact the performance of your application. This might be a factor to consider when designing your logical model.

For example, we record prices from a historical price table for each product. When the line items are added, the cost is not written to the line-item table, but rather extracted from the historical table based on the date the order was place. When we go to compute the total cost of the order, we need to:

- Get all the items on the order.
- For each item, look up the product code to get price, based on the date of the order.
- Sum up all costs.

This process could be slow, particularly for a database with a large historical price table or frequently large number of items ordered. If a monthly or yearly report is requested, showing orders and total cost, it could take a while to recompute the cost for each order.

You might want to denormalize the line-item table and store the actual cost when the order is placed. This is redundant (since cost is a lookup value) but might improve performance for a large report. However, there is a risk that if the historical price is updated, the line-item table (now containing a cost column) will not pick up the updated price.

While denormalizing is a risk, it might be acceptable based on your application. You should always consider the benefit versus risk. If your application is selling high-end items, and a price variation could be in the thousands of dollars, you might not want to take the risk. However, if the price variation is only a few cents, the performance gain would likely outweigh the risk of a price discrepancy of a few cents.

Summary

The process of normalization reduces the likelihood that data is duplicated while maintaining flexible data sets. The first three normal forms are typically the level you should aim your logical model at. The additional forms handle some unusual but possible situations that could occur in your database.

Chapter 5 Physical Data Model

The physical model takes the logical model and defines items like the database tables, fields, and indexes necessary to store the entities of the model in a particular database management system. Typically, a database developer or database administrator should build the physical model since they likely are familiar with the nuances of the target database.

Tables

A table is a collection of fields that represent attributes from the logical model. Typically, the table will contain a primary key (PK), a way to uniquely identify each object in the table. The logical model attribute can be a single field, or a collection of fields, which together form the PK attribute. The primary key can be one or more columns, as needed to identify each record uniquely. Your logical model should indicate how the primary key is constructed.

Naming

There are few conventions for naming tables, although people are often opinionated about them. If your team has table naming conventions, you should adhere to them. If there are no naming conventions, consider defining them. A few suggestions follow.

Use singular form

This is a matter of preference, and other developers might suggestion plural table names. I like singular form because I find the syntax: **Customer.Address** clearer than **Customers.Address**. A table implies a collection of objects, so the plural form is somewhat redundant. Also, particularly in English, plural forms are not consistent. **Customer** becomes **Customers**, but **Person** becomes **People**, **Activity** become **Activities**, and **Data** remains **Data**.

Avoid database keywords

One convention that most developers agree on is not to use database keywords as table names. SQL Server for example, requires delimiters (brackets []) around table names that are keywords. mySQL uses backticks (`) around reserved words. There is no need to place a burden on anyone referencing the table to (a.) know the keywords, and (b.) know what delimiter should be used.

Be consistent with naming

If you have tables with similar content, try to keep the names similar. Calling the vendor table **Vendor_location**, and the customer table **Customer_Shipping_Address** is not a good idea if both tables contain address information associated with a company (vendor or customer).

Fields

The field is the smallest unit in the database, and all attributes will become one or more fields in the table. Each field will have a unique name within the table, a data type (with an optional size value), and an indication of whether the field can be empty (NULL value).

When defining a field, you should adhere to a few guidelines:

- Keep the field name the same across tables. If you decide to use **POSTAL_CODE**, don't use **ZIP** or **ZIPCODE** in another table.
- Select the smallest appropriate data type for the field.
- Always define **NULL** or **NOT NULL**, even if the database provides a default.
- Try to use **CONSTRAINTS** and **DEFAULT** values to ensure the quality of the data in the table.

When a database system retrieves a record, it copies the record into a memory buffer. The smaller the total record size, the more records can be placed into a buffer. If a query returns more than one record (likely), then the less memory needed for all the records, the better the performance will be. Also, many database systems will hold records in a cache, anticipating the records might be needed again. The more records that can be retrieved from the cache rather than by accessing the disk, the faster the query will perform.

Data types

There are four major data type categories that can be used in a SQL table. These are string, numeric, date/time, and Boolean (bit) value. Within each category, there typically are more detailed variations, such as size, range of dates, integer versus decimal, and so on.

In this section, we will describe Microsoft SQL Server data types. Appendix A lists the data types for other SQL versions (Oracle, mySQL).

String data types

The string types are referred to as CHAR data. The general syntax to define a character/string field is the following.

```
FieldName       CHAR (size) NULL | NOT NULL
```

When defining a string type, there are a few considerations.

The **CHAR(size)** is a fixed size, it will always consist of the specified size characters. If the data content is shorter than the size, it will be padded out (typically with spaces).

The **VARCHAR(size)** is a variable sized string field, with a maximum of **size** characters.

If your field will always be the same size (such as two-character United States state code), **CHAR** data should be used. If, however, the content placed in the field varies substantially, you should use the **VARCHAR** data type. The **VARCHAR** data type places a bit more work on the server, since it needs to store an integer indicating how much data is in the field.

Note: In Microsoft SQL Server, the `size` parameter is optional, but if not provided, it defaults to 1. However, if you use `CAST` or `CONVERT` and don't specify a size parameter, the default size is 30.

If you try to put more content than the size supports, you will receive an error message to the effect that data will be truncated, and the field will not be updated. Also, if an application code (say C#) retrieves the data from **CHAR** data, it will not be trimmed and might contain trailing spaces. If the application retrieves data from **VARCHAR**, it will be not have trailing spaces.

When deciding between **CHAR** and **VARCHAR**, you need to consider the tradeoffs. Since SQL knows the size of **CHAR** fields, it is easier to move them into a memory buffer. With a **VARCHAR** field, SQL needs to determine the size first, before adding it to the memory buffer. Using **VARCHAR** fields makes sense for many applications because the performance hit is measured in milliseconds. If you are consistently returning many records and performance is a top consideration, a **CHAR** field might be better.

The keyword **MAX** can be used in place of a size value, and typically would be used when fields vary greatly and could possibly exceed 8,000 characters.

Unicode characters

The **CHAR** data typically holds ASCII characters, which are characters that can be represented by a single 8-bit value (1-256). However, if you are dealing with international data, it is possible that the data will be in Unicode (two bytes are needed to represent each character). If you anticipate UNICODE data, you can use **NCHAR()** or **NVARCHAR()**. In SQL Server, the **LEN()** function will return the number of characters in the field, while the **DataLength()** function will return the actual number of bytes used to hold the content.

Collation

In a simple world, all data could be stored in ASCII, and the numeric bytes would correspond to the same letter or symbol. However, SQL is used worldwide, and different countries and cultures use different alphabets and symbols. SQL lets you specify the set of characters and symbols (called a collation) to use. You can specify the collation at a server level, a database level, and even an individual field level.

To add a specific collation on a column, you simply add the keyword **COLLATE**, followed by the collation name. For example, the following.

```
Notes NVARCHAR(max) COLLATE SQL_Latin1_General_CP1254_CI_AS
```

This syntax would allow the notes to be entered using Turkish characters. In addition, the sort order is determined by the collation. The collation provides details, for example, of how to sort the Turkish characters ç, ü, and û.

You can find the collations on your server using the SQL command.

```
SELECT * FROM sys.fn_helpcollations()
```

```
WHERE name LIKE 'SQL%'
```

The following command will show you the collation of the server and database.

```
SELECT 'SERVER' as type,
        CONVERT (varchar(256), SERVERPROPERTY('collation')) as Collation
UNION

SELECT 'DATABASE',

        CONVERT (varchar(256), DATABASEPROPERTYEX(db_name(),'collation'));
```

When to use character types

A character data type imposes the least restrictions on the type of data that can be entered, so you should keep that in mind when choosing the type. Some developers will simply use **CHAR** types for all fields, which is particularly common for date values. However, often a direct result of using **CHAR** fields for dates is getting an error message when trying to convert the string to a date. In SQL Server, the syntax **IsDate('2021')** returns 1 (true), and will refer to 1/1/2021. If you anticipate doing any kind of operation on the field (such as adding days), you should use a better data type than **CHAR** data.

Numeric types

For numeric types, there are integer values and decimal values. The integer values are always exact values, and the type of integer (**TINY**, **BIG**, and so on) simply refers to the range of values that can be placed in the field:

- **Bit**: A Bit field can be 0, 1 or NULL (this is often used in place of the Boolean data type that SQL does not support).
- **TinyInt**: A whole number between 0 and 255.
- **SmallInt**: A whole number between -32,768 and 32,767.
- **Int**: A whole number between -2,147,483,648 and 2,147,483,647.
- **BigInt**: Whole number between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807.

Note: `TinyInt` does not allow negative numbers; you will receive an overflow message if you try to place a negative number into a `TinyInt`. (Similarly, you'd receive the same error message if you tried to set an integer value outside the allowed range.)

Working with integer data is generally quicker than decimal values.

Decimal (numeric)

The decimal or numeric data type (they are functionally synonyms) are exact decimal values. You will need to specify the precision (total number of digits) and the scale (digits to the right of the decimal points). The maximum precision is 38 digits, and the default precision is 18 digits.

The scale portion can be zero up to the precision specified. You cannot specify the scale without also specifying the precision.

If you try to place a number larger than the precision into the field, you will get an overflow error message. If you try to place a larger number into the scale, it will be rounded up or down.

Table 12 shows example values using a field with a **`Decimal`** (5,2) data type.

Table 12: Example numeric storage

Value	Stored	Notes
123.45	123.45	Stored exactly
234.251	234.25	Rounded down
234.489	234.49	Rounded up
1234.50	Nothing	Overflow error message

Float and real

Floating point numbers are approximate values, which use less storage space than decimal values but don't provide precise values. The floating data type specifies the size of the mantissa (which defines the number of bits to store). The mantissa value specified can be any number from 1 to 53, although SQL only uses 24 or 53 (numbers less than or equal to 24 are assigned size 24, and any number above that is assigned 53).

A double precision float (**`Float(53)`**) can hold $1.8 * 10^{308}$ (which is a lot of zeros). The **`real`** data type is a synonym for **`Float(24)`**. If you are concerned with storage space, consider using a floating-point number, but be aware of the loss of precision that can occur.

Money types

SQL Server has two money types: **`smallmoney`** and **`money`**. The money types are accurate to a ten-thousandth of the monetary units. A period is used to separate the fraction (like cents) from the whole value (like dollars). The money data type can be many different currencies that support whole and fractional units, such as U.S. dollars, British pounds, or Japanese yen.

The **`smallmoney`** type can range between -214,748.3648 and 214,748.3647. The **`money`** data type can range between -922,337,203,685,477.5808 and 922,337,203,685,477.5807. **`money`** requires 8 bytes; **`smallmoney`** requires 4 bytes of storage.

Note that money types are based on **`INT`** and **`BIGINT`**, but the types support the decimal place splitting between whole and fractional currency units.

Date and time

SQL Server provides several different date and time data types, depending on the date range needed, when to include time, and the time zone offset:

- **Date**: The **Date** type holds only date values (no time portion) between 1/1/0001 and 12/31/9999. It is accurate to a single day, and defaults to 1/1/1900.
- **DateTime**: The **DateTime** type holds date values between 1/1/1753 and 12/31/9999 and includes a time portion between 00:00:00 and 23:59:59:997. It defaults to midnight, January 1 of 1900.
- **DateTime2**: This type is very similar to **DateTime**, with a wider range of dates (1/1/0001 through 12/31/9999) and more precise time (00:00:00 through 23:59:59:99999999).
- **SmallDateTime**: This is a version of date-time that uses less storage by only supporting dates of 1/1/1900 through 6/6/2079 and time down to the second (23:59:59).
- **Time**: Just the time portion with a range of 00:00:00:00000000 to 23:59:59:99999999.
- **DatetimeOffset**: This is just like **DateTime2** (1/1/0001 through 12/31/9999), with the addition of the time zone offset (number of hours different from UTC time).

If you plan on any date manipulation, use the date and time data types. A very common question on programming support boards deals with dates being stored in characters fields.

Boolean

SQL Server does not provide a Boolean data type, but the numeric **BIT** type can hold a 0, 1, or NULL. You could define a column as a **BIT** field, **NOT NULL**, to create a true Boolean value.

Constraints

A constraint can be added to the field definition, and this will control the type of data allowed in the field. Each database will have its own allowed constraints, but four common constraints are the following.

Primary key

The column (or columns) that forms the table's primary key. By design, this column must contain unique values, since a primary key is the way to uniquely identify a row. Attempting to duplicate a primary key value will result in an error. In addition, a primary key cannot be NULL.

You can specify a primary key by adding the **PRIMARY KEY** keyword after the column definition.

```
ID int NOT NULL PRIMARY KEY
```

Alternately, you can do it by creating the constraint after the columns are defined.

```
PRIMARY KEY (id)
```

You should use the **CONSTRAINT** syntax to create a primary key on more than a single column, for example, **ORDER_NO** and **LINE_ITEM_NO**.

Foreign key

The column in the table must be NULL or match another table's primary key. This is how table relationships are built. To create the foreign key, you will need to know the table and primary key column for the relationship.

To create the foreign key on the column definition, you can use the foreign key syntax.

```
LocationID int FOREIGN KEY REFERENCES ShippingLocations(LocationID)
```

You can create the foreign key after the columns are defined (which is necessary if more than one column is the key) using the FOREIGN KEY constraint syntax.

```
FOREIGN KEY (CustNo, LocaNo) REFERENCES Shipping(CustNo,LocaNo)
```

The reference should specify a table and its primary key (one or more columns).

Unique constraint

The unique constraint requires unique values for a column or columns within the table. If you are using a surrogate primary key (for internal storage) and a visible key for the user, be sure to add the unique constraint for the visible key.

A unique constraint can also be applied to a combination of fields. For example, we might want a constraint that user ID and email together must be unique. This would prevent a user from creating multiple accounts with the same user ID and a single email address.

You can add the keyword **UNIQUE** after a column definition to specify the column is unique.

If you need a unique set of columns, you will need to use the **UNIQUE** constraint, as shown in the following example.

```
UNIQUE (UserId, EmailAddress)
```

If you place the **UNIQUE** keyword after each field, you will require that each field individually must be unique, not the combination of the two fields.

Check constraint

A check constraint is a logical expression that must return **TRUE** for the value to be allowed in the column. For example, you might want to create a constraint that the due date of an invoice is not in the past.

```
Check (Inv_Due_Date >= GetDate())
```

Developers sometimes skimp on constraints, which can likely cause problems down the road. For example, if a secondary key is used to identity the record (such as customer ID when the primary key is the customer record number), any code that references the customer ID probably expects a single row to be returned. If a unique constraint was not added and a customer ID gets duplicated, it will very likely break the code that never anticipated duplicate customer IDs.

Similarly, imagine a birth date field that allows future dates. What will the application code do when it computes a negative age for a person?

It is generally a good practice to constrain the data to prevent unexpected errors in the future.

Defaults

Defaults are field values that get placed into record if the field is **NULL** during an **INSERT** operation. Defaults only apply during an **INSERT** command. If your **INSERT** statement provides a value for the column, the default will not be used.

A default can be a constant (text or numeric value) or an expression that returns a constant value. You can also use some SQL variables, such as **@@ServerName**, and functions, such as **db_name()** or **getDate()**. **getDate()** is often used to default a column to the current date.

The **DEFAULT** keyword, followed by the value, can be specified on any column for which you wish to add a default value.

Note: DEFAULT only applies if you don't specify the column name during the INSERT. If you specify the column name and pass it a NULL, the NULL value will be written to the column, not the DEFAULT value.

Computed fields

A computed column is a virtual column where the expression to compute the value is stored in the table, not the actual value.

To create the column, you provide a column name, the keyword **AS**, and a SQL expression. Whatever data type the expression returns will be the data type of the column. For example, the following syntax would return the total value of an inventory item based on quantity on hand and per unit cost.

```
Total_Value AS QtyOnHand * PerUnitCost
```

While this is a simple example, and could be computed in a front-end, if the formula was complex and used in multiple places, having the definition in the database can make sure the value is consistent across applications accessing it.

Note: Computed columns that are complex can slow down query performance, since the expression needs to be evaluated for every row. SQL Server allows you to add a PERSISTED option to the definition, meaning the value is saved when the row is INSERTED. It will be updated if the data is changed, but when just querying the data, the current persisted value will be shown, not recomputed.

Logical entities to physical table

For an example, let's consider the customer entity, which we normalized in the prior chapter. The logical model is duplicated in Figure 19.

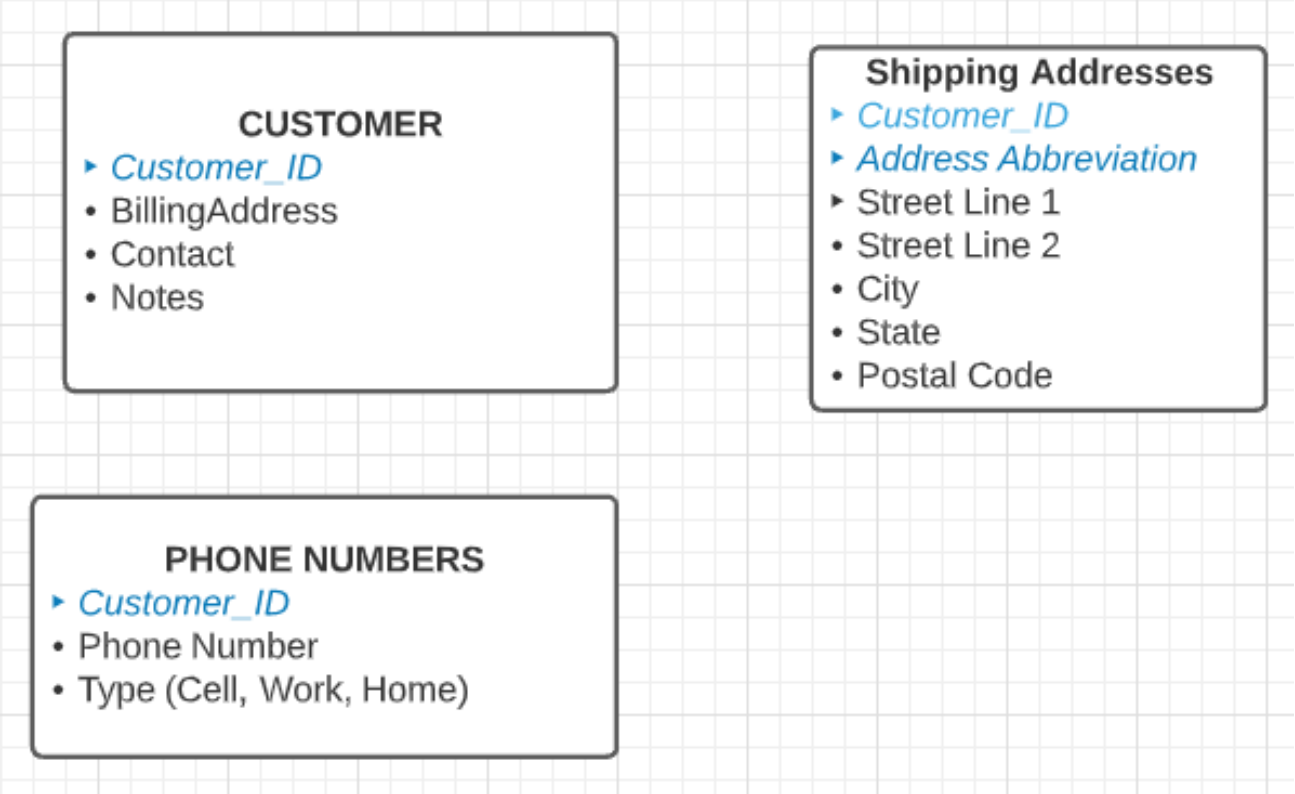


Figure 19: Customer logical model

Customer table

For the customer table, we need to select a primary key. There are a few different approaches we can use.

Customer ID

We can create an identity column and use this as the key. The customer will never see the key, but it will be used to link the table together. This approach allows the system to maintain the key and prevent duplicate key values.

> ***Tip: Often, a website might accept parameters to display customer information. If the parameter is an integer, a hacker might simply modify the URL and change the customer ID parameter, possibly viewing other customer data. Keep this in mind if you use a predictable pattern to assign ID values.***

Another approach is like the identity column, but uses a GUID (globally unique identifier) as a primary key (using the **`NewId()`** function as a default for the Customer ID column). This will prevent hackers from viewing other records but will be less convenient as a parameter to a web URL.

We can create a key based on the customer's name so the customer can easily remember the key. While this will make the key easier to understand, it also would be updated if the customer's name changes.

For our customer table from the model, we are going to use the identity column for the key to link tables together and provide a different identifier for the customer to use.

Our final table design might look like this.

Table 13: Customer table

Field	Type	Nullable	Default	Constraint
Customer_ID	**INT**	**NO**	Identity	None (implied no duplicates)
CustomerCode	**varchar(15)**	**NO**		No duplicates
CompanyName	**varchar(50)**	**NO**		
Address1	**varchar(75)**	**NO**		
Address2	**varchar(75)**	**YES**		
City	**varchar(50)**	**NO**		
StateCode	**char(2)**	**NO**		
Postal_Code	**varchar(10)**	**NO**		**LEN(Postal_Code)=5 OR LEN(Postal_Code)=10**
DateCreated	**Date**	**NO**	**GetDate( )**	
Contact	**varchar(75)**	**YES**		
NOTES	**varchar(max)**	**YES**		

We defined the internal **customer_ID** to be an identity, surrogate key, and the **CustomerCode** to be the customer visible code for their company. We also require **Postal_Code** to be either five characters long or 10 (if company uses zip+4). Note that this example only considers addresses in the United States.

The **Notes** field is set to **varchar(max)**, which allows unlimited notes to be added. However, the notes are assumed to be English or other European languages. If we want to allow foreign text, we can use the **nvarchar()** data type instead.

Figure 20 shows the SQL Server transact-SQL code to create the table.

```
CREATE TABLE dbo.customer
(
    Customer_ID     int IDENTITY PRIMARY KEY,
    CustomerCode    varchar(15) NOT NULL UNIQUE,
    CompanyName     varchar(50) NOT NULL,
    Address1        varchar(75) NOT NULL,
    Address2        varchar(75) NULL,
    City            varchar(50) NOT NULL,
    StateCode       char(2) NOT NULL,
    Postal_Code     varchar(10) NOT NULL CHECK( LEN(Postal_Code)=5 or LEN(Postal_Code)=10),
    DateCreated     Date        NOT NULL DEFAULT getDate(),
    Contact         varchar(75) NULL,
    NOTES           varchar(max) NULL
)
```

Figure 20: Create customer table

SQL Server provides certain extended properties about the column in the table. A useful property is a column description field. You can add customer descriptions to a table using the **sp_AddExtendedProperty** stored procedure. For example, Figure 21 shows how to add a description property to the **CustomerCode** for this table.

```
exec sp_AddExtendedProperty
       'MS_Description'
     , 'Customer code is external name customers use to identify themselves'
     , 'SCHEMA', 'dbo'
     , 'TABLE', 'Customer'
     , 'COLUMN', 'CustomerCode'
```

Figure 21: Add Description property

Since the **MS_Description** property is not standard, you will need to use the system tables to get the description. Code Listing 1 shows the code to display the column name, type, and description for a table.

Code Listing 1: Show column properties

```
SELECT sc.name AS column_name, st.name AS dataType,
      IsNull(pt.value,'') AS Description
      FROM syscolumns sc
      JOIN systypes st ON st.xtype=sc.xtype
      LEFT JOIN sys.extended_properties pt
                ON pt.major_id=sc.id AND pt.minor_id=sc.colid
      WHERE object_name(id)='Customer'
      ORDER BY sc.colorder
```

The **Description** property can be a handy documentation component when building your database, and it will be maintained within the database for future reference. You can also set a table-level property by not providing a column name in the stored procedure.

Address table

The address table is used to provide the alternate shipping locations that a customer might have. The table design might look like this.

Table 14: Address table

Field	Type	Nullable	Default	Constraint
Customer_ID	**INT**	**NO**		**PRIMARY KEY** (Also foreign key to **Customer**)
AddrAbbr	**varchar(15)**	**NO**		**PRIMARY KEY**
Address1	**varchar(75)**	**NO**		
Address2	**varchar(75)**	**YES**		
City	**varchar(50)**	**NO**		
StateCode	**char(2)**	**NO**		
Postal_Code	**varchar(10)**	**NO**		**LEN(Postal_Code)=5 OR LEN(Postal_Code)=10**
PreferredShipper	**varchar(12)**	**NO**	**FEDEX**	**CHECK(preferredShipper IN ('FEDEX','UPS','FRIEGHT'))**
ShippingCost	**Computed**			Varies by shipper
NOTES	**varchar(max)**	**YES**		

We would add the following constraints to this table.

```
PRIMARY KEY (Customer_ID,AddrAbbr)

FOREIGN KEY (Customer_ID) REFERENCES Customer(Customer_ID)
```

When we create the order table, we will have a foreign key relationship to the address table, so the warehouse knows to which address to ship the products and the preferred shipper to use.

Figure 22 shows the code to create the shipping address table.

```
CREATE TABLE dbo.Shipping_Address
(
    Customer_ID         int NOT NULL,
    AddrAbbr            varchar(15) NOT NULL,
    Address1            varchar(75) NOT NULL,
    Address2            varchar(75) NULL,
    City                varchar(50) NOT NULL,
    StateCode           char(2) NOT NULL,
    Postal_Code         varchar(10) NOT NULL
                        CHECK( LEN(Postal_Code)=5 or LEN(Postal_Code)=10),
    PreferredShipper    varchar(12) NOT NULL DEFAULT 'FEDEX'
                        CHECK( PreferredShipper in ('FEDEX','UPS','FREIGHT') ),
    ShippingCost        AS
                        CASE
                         WHEN PreferredShipper = 'FEDEX' THEN 50.00
                         WHEN PreferredShipper = 'UPS' THEN 20.00
                         ELSE 100.00
                         END,
    NOTES               varchar(max) NULL
    PRIMARY KEY (Customer_ID,AddrAbbr)
    FOREIGN KEY (Customer_ID) REFERENCES Customer(Customer_ID)
)
```

Figure 22: Create shipping address table

After a few descriptions and running the query in Code Listing 1, we can see the table details in Table 15.

Table 15: Documentation for shipping address table

column_name	dataType	Description
Customer_ID	int	PK (with AddrAbbr)
AddrAbbr	varchar	A short abbreviation for this shipping address
Address1	varchar	
Address2	varchar	
City	varchar	
StateCode	char	
Postal_Code	varchar	
PreferredShipper	varchar	
ShippingCost	numeric	Computed cost based on preferred shipper choice
NOTES	varchar	

Phone number table

The phone number table is simply a list of phone numbers and types associated with the customer. As currently described, it is simply a list of phone numbers within a key to identify individual phone numbers. While this logical design is technically accurate, I would suggest we always add a primary key (surrogate key in this case).

With this tweak, our phone number table could look like this.

Table 16: Phone number table

Field	Type	Nullable	Default	Constraint
PhoneNumberID	**INT**	**NO**		**PRIMARY KEY**
Customer_ID	**INT**	**NO**		**FOREIGN KEY** to customer
PhoneType	**char(4)**	**YES**	**WORK**	**CHECK** (**PhoneType** in ('**HOME**','**WORK**','**CELL**'))
PhoneNumber	**varchar(32)**	**NO**		

The reason I suggest the surrogate key is to make the rows easier to update and delete. In addition, at some future point, you will likely be asked to be able to identify a particular phone number from the list.

Figure 23 shows the code to create the phone numbers table.

```
CREATE TABLE dbo.Phone_Numbers
(
    ID          int NOT NULL PRIMARY KEY,
    Customer_ID int NOT NULL,
    PhoneType   char(4) NULL DEFAULT 'WORK'
                CHECK( PhoneType in ('WORK','CELL','HOME') ),
    PhoneNumber varchar(32) NOT NULL,

    FOREIGN KEY (Customer_ID) REFERENCES Customer(Customer_ID)
)
```

Figure 23: Create phone numbers table

Indexes

SQL can create an index to improve performance of the statement to retrieve data. An index is like a book index, a keyword or keywords, and a record number where it can be found. Imagine a phone book (an ancient book containing an alphabetical list of names and phone numbers). You would likely open the book to the middle and see what letter the names begin with (then either open the book to another page higher or lower until you find the name). This is a binary search, which is a very simplified explanation of how indexes work.

SQL will create indexes on primary keys automatically, but you can decide what other indexes would be useful. In our phone book example, imagine you need to find a phone number, but you only know the street name. Starting at page one and looking for the street would be very slow (in SQL, this is called a table scan). Creating an index on the street name (keyword and page) would allow you to find the phone number by street name much quicker.

While SQL will allow you to create any number of indexes (subject to database limits), there is a performance cost when adding to or editing the database, since the index likely needs to be updated, as well. You need to consider your queries and consider adding indexes for the very commonly looked up fields. In our customer table, the **CUSTOMER_ID** field will automatically have an index, since it is the primary key. You might want to add a second index on the customer code, since that is most likely how the various departments will look up the customer information.

With SQL Server and other database systems, you can often profile your queries to pick up suggested indexes if a query is running slow. The syntax to create an index is the following.

```
CREATE INDEX <name> ON <table (field or fields)>
```

We could create the customer code index as follows.

```
CREATE INDEX byCustCode ON Customer(CustomerCode)
```

Determining which indexes to create is often a matter of trial and error to improve query performance. Very small tables (lookup tables) often only need the primary key index. Large, user-facing tables generally have multiple indexes based on the end users' use of the system.

Views

One of the minor drawbacks of data normalization is that some simple queries become more complex when you need to join tables together. For example, if our employee table supports multiple phone numbers for employees, and we wanted an internal phone number list, our query could look like the following snippet.

```
SELECT emp.first_name,emp.last_name,

       IsNull(pho.PhoneNumber,'<none>' as WorkPhone

FROM employee emp
```

```
LEFT JOIN phone_numbers pho on pho.EmployeeId=emp.ID and Phonetype='WORK'
```

A person who is less familiar with SQL might not understand that query, so we can create a view to make it simpler to reference the phone list.

```
CREATE VIEW work_phone_list

AS

      SELECT emp.first_name,emp.last_name,

            IsNull(pho.PhoneNumber,'<none>' as WorkPhone

      FROM employee emp

      LEFT JOIN phone_numbers pho on pho.EmployeeId=emp.ID

            AND Phonetype='WORK'
```

Now the user simply can write the following.

```
SELECT * FROM work_phone_list
```

The complexity of the **JOIN** query is hidden from the user. We can also use this approach to return only a subset of fields. If an employee record included salary information, we might want to create a view for Human Resources that only returns columns other than salary information.

Note: Views cannot include an `ORDER BY` clause in most database systems, although there are ways to work around it, depending on the SQL system used.

Summary

We provided a high-level overview of how to create a physical model in this chapter using Microsoft SQL Server. Every database package has its own enhancements and nuances, possibly additional field types, table features, and so on. Once you have the general table and field definitions spelled out, you can work with the database administrators to best create your tables using the best features from the database package.

Chapter 6 Data Standards

While modeling a database, you should strive to use standards for common attributes, such as addresses, countries, and phone numbers. Doing so will allow other developers to understand your model better and will likely increase your model's robustness. You can develop your own standard naming conventions, field lengths, data types, and so on. Or you can take advantage of existing standards to apply. In this chapter, we will review some of the common public standards that might help when building your database.

The ISO (International Organization for Standardization) develops and publishes standards for almost everything, such as countries, languages, and phone numbers. If an ISO standard exists, it would be beneficial for your database to use it. This allows other developers to comfortably work with your data and facilitates easier data exchange.

In addition, many government agencies also define standards. If your application needs to interact with a government agency, it would be beneficial to use some of those standards as well. Even if your application isn't used in government, using the standards can still be helpful for exchanging data with other organizations.

Internal standards

When designing your database, you should define standard values for certain column types. For example, the United States Social Security number (SSN) might be defined as characters, in groups of three digits, a dash, two digits, a dash, and four digits.

You might want to ensure, for example, that state or province codes are always two characters, and your city fields are always 30 characters long. Fortunately, Microsoft SQL Server supports the concept of user-defined data types. These are named data types that can be used to provide column size, defaults, null option, and so on. By using these types consistently, you can keep your column consistent across the tables.

User defined type

To create a user defined type, use the following syntax.

```
CREATE TYPE [schema].[type_name]

FROM base_type

NULL | NOT NULL
```

The base type is a standard SQL data type and optional size. You can also indicate whether the field accepts NULL values.

To create an SSN type, we could use the following example.

```
CREATE TYPE dbo.SSN

FROM

varchar(11) NOT NULL
```

We can add a check constraint to the type as well. The check constraint must be defined as a rule within SQL Server, and that rule would be bound to the data type. For example, we want to enforce the format of the Social Security number (according to U.S. usage).

Create a rule

A rule in SQL Server is a general-purpose, conditional expression. You create the rule with the following syntax.

```
CREATE RULE [schema].[rule_name] AS <conditional_expression>
```

The rule will have one variable associated with it, and it will contain the **INSERTED** or **UPDATED** value when the rule is applied. Our rule for the **SSN** field might look like the following code snippet.

```
CREATE RULE dbo.SSNCheck
AS
@val LIKE '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]'
```

This rule says that the value (**@val**) of the field must be three digits, a dash, two digits, another dash, and four digits. The rule is valid if the conditional expression returns **TRUE**.

Once the rule is created, you can bind it to the data type using the **SP_bindrule** stored procedure.

```
EXEC SP_bindrule 'SSNCheck', 'SSN'
```

Create a default

SQL Server also allows you to create a **DEFAULT** rule, which can be applied to a matching column type. For example, let's assume you always want to default to the state your company is located in. You might create a simple default expression, such as the following.

```
CREATE DEFAULT stateCode AS 'PA'
```

You can then bind that default value to a user-defined data type, as well.

SQL Server SSMS has a dialog box (as shown in Figure 24) that allows you to create user-defined types, rather than relying on SQL scripts.

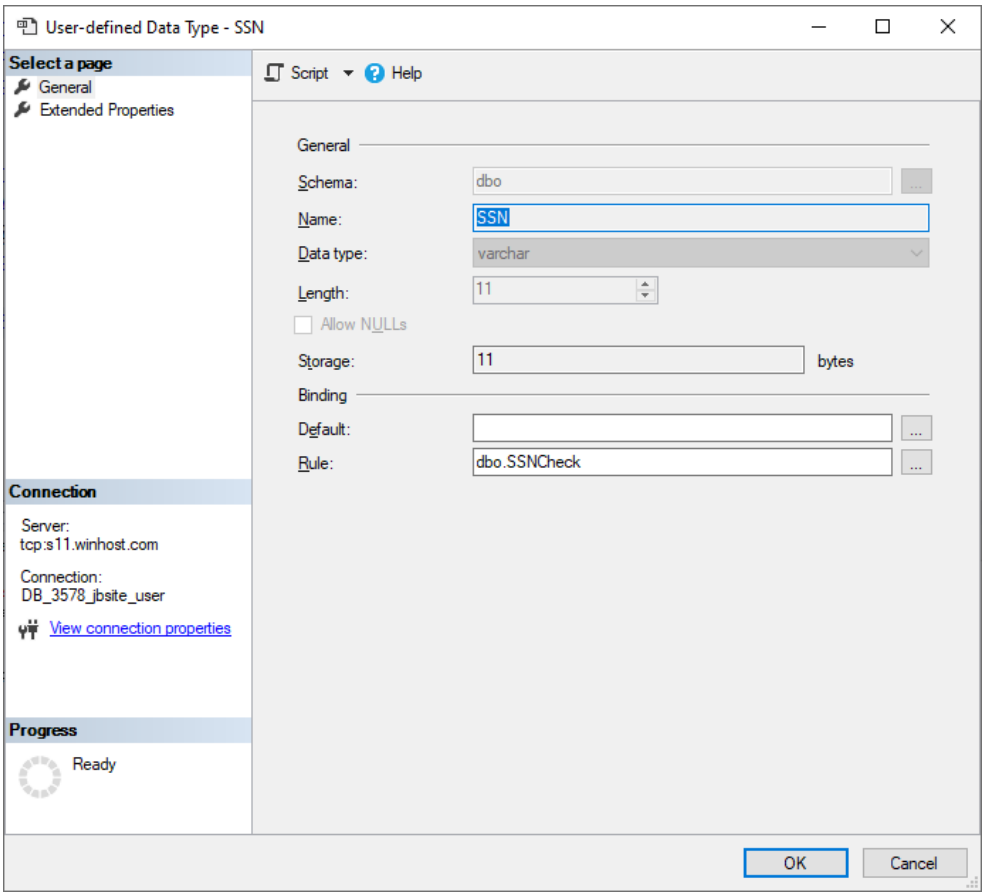


Figure 24: Dialog box to create user-defined types

By creating user-defined data types and documenting them, you can ensure that the tables within your database have consistent type, sizes, nullable status, and so on.

User-defined table types

In addition to column user-defined types, you can also create user-defined table types. A user-defined table type is created using the following syntax.

```
CREATE TYPE <name> as TABLE
(
  Column Definitions
)
```

The column definitions are the same column definitions that **`CREATE TABLE`** uses. For example, you might want to create a table type to be used for lookup tables. One drawback is that you cannot create a new table based on the type, but rather can create a table variable. This would be useful for stored procedures that need temporary tables, and you'd prefer them to be consistently defined.

Note: There are workarounds to create a table from the user-defined table type; however, these solutions typically don't pick up constraints, defaults, and so on. The following code snippet shows how to create a table based on a user-defined table type.

```
DECLARE @table AS dbo.UserDefinedTableName

SLECT * INTO dbo.DesiredTable FROM @table WHERE 1=2
```

This will create the table with the structure, but any constraints or defaults will not be present in the newly created table.

External standards

While defining user types can provide internal consistency in your table design, you should also strive to rely on existing standards for names, data types or sizes, and contents. For example, if you want to keep track of the type of business for a customer, rather than create your own coding scheme, you might want to look at SIC (Standard Industrial Classification) so if you exchange data with an external source, they will likely understand your data values.

ISO standards

The ISO (International Organization for Standardization) was founded in 1947 and is currently headquartered in Geneva, Switzerland. They create and certify standards that are defined by international experts. The standards that are developed are based on the consensus of the experts in the field, and all comments are considered before the standard is certified.

Country codes

ISO Standard 3166 provides two- and three-letter codes, as well as numeric values to represent countries of the world. A few example values are shown in Table 17. By using either the code or numeric values, other databases can interpret your database columns as countries.

Table 17: Sample ISO-3166 country information

English name	Alpha-2	Alpha-3	Numeric
Belize	BZ	BLZ	084
Canada	CA	CAN	124
Germany	DE	DEU	276

English name	Alpha-2	Alpha-3	Numeric
United States	US	USA	840

You can browse the country codes on the ISO website. Search for **3166** and navigate to the country code list.

Currency codes

ISO Standard 4217 provides three-letter and numeric codes to represent various currencies of the world. A few example values are shown in Table 18. By using either the code or numeric values, other databases can interpret your currency type.

Table 18: Sample ISO-4217 currency information

Currency	Code	Numeric
Canadian Dollar	CAN	124
Swiss Franc	CHF	756
Japanese Yen	JPY	392
United States Dollar	USD	840

You can browse the country code on the ISO website. Search for **4217** and navigate to the currency list.

Language codes

ISO 639 provides codes for languages used throughout the world. Table 19 shows sample language data from ISO 639.

Table 19: Sample ISO-639 language codes

Language name	Native name	Code-2	Code-3
Chinese	中文 (Zhōngwén), 汉语, 漢語	zh	chi
French	Français	fr	fre
Greek	ελληνικά	el	gre
English	English	en	eng

Note that there are three versions of the three-character code (standards 639-2/T, 639-2/B, and 639-3). Be sure to communicate which version you are using when interfacing with external applications.

The country, language, and currency code standards are freely available from ISO. Most of the other standards are copyright protected. If you plan on using other standards from ISO, be sure to adhere to the appropriate copyright laws.

Wikipedia provides documentation and a good visual representation on these free standards:

- ISO 3166
- ISO 4217
- ISO 639

You can also download the standards at many different websites. I would suggest searching GitHub for downloadable files.

Phone number standards

ISO does not publish phone number standards. However, the ITU (International Telecommunications Union) has created a standard called E.164. It basically breaks a phone number into two components: a 1–4-digit country code followed by the phone number itself. Based on this, we might want to break the phone field into two fields (country code and number):

- **CCC varchar(4)**
- **PhoneNumber varchar(15)**

One approach to deal with phone numbers is to store just the digits in the database field, and then format the number based on the country code. Code Listing 2 shows a sample SQL computed field to format the phone number field based on the ISO country code.

Code Listing 2: Computed phone number display

```
create table dbo.SamplePhone
( ISOCode      char(2),
  phoneNumber  varchar(15),
  formattedPhone
  as
  (case
   -- United States
   when ISOCode ='US'
then '+01 ('+substring(phoneNumber,1,3)+') '+
                  substring(phonenumber,4,3)+'-'+
                  substring(phonenumber,7,15)
   -- Mexico
   when ISOCode ='MX'
      then '+52 ('+substring(phoneNumber,1,2)+') '+
                  substring(phoneNumber,3,4)+' '+
                  substring(phoneNumber,7,4)+' '
   -- Germany
   when ISOCode ='DE'
```

```
      then '+49 '+substring(phoneNumber,1,3)+' '+
                  substring(phonenumber,4,15)
   else phonenumber
   end
   )
)
```

Note: Some database purists might rightfully point out that the formatting of the phone number could be done in the front-end. This is one approach; however, for a large collection of applications, it is possible that not all developers will approach formatting the same way. Having the database system format the number at least ensures it will be display consistently across various applications.

Address standards

No official organization publishes address standards since they vary so much by country. It is one of those thorny data processing issues that has many third-party solutions, but no defined standard. There are multiple approaches to handling addresses in the database.

Be lazy

With this approach, you simply add 3-4 lines called **`Address_Line_X`**, set them to a reasonable size, and let the application users determine the best way to write out the address. While it can work, it introduces a few problems:

- There's no easy way to identify duplicate addresses.
- A lot of expectation is placed on the application users.
- Reporting is difficult, and determining a state code consistently would be very hard to do.

Pick a country

If all your addresses are within a single country, it becomes easier. Most country post offices and government agencies define what address formats should look like. For the good news, the United States Postal Service provides documentation on addressing standards.

The bad news is the documentation is over 200 pages long. In general, the USPS format is 4–5 lines:

- Line 1: Information/attention
- Line 2: Company name
- Line 3: Delivery address
- Line 4: City, state, and zip code
- Line 5: Country (only if outside U.S.)

The post office has considerable flexibility in handling addresses. The delivery address could be a P.O. Box, a rural route, a number, and street name. Mail can be delivered even without the zip code.

If you need to determine other country address formats, consider BitBoost Systems. They allow you to get formatting information for many countries. You can also consider the Universal Postal Union (a United Nations division). They provide address formatting information for most member nations.

If you know you'll only mail to one country, determine the format, and create individual fields to hold the elements, such as city, state, and postal code.

Multiple countries

If you must support mailing to multiple countries, the concept is the same as a single country, except you need to make sure you have fields for all possible address components. You will also likely need to loosen validations on the postal code field, since postal codes vary by country.

For sake of example, let's assume you've added the following fields:

- Contact
- Number
- Street
- City
- Postal
- State

I would then suggest creating a table of address templates, such as those shown in Table 20.

Table 20: Sample address templates

Country	Line 1	Line 2	Line 3
Mexico	~Contact~	~Street~ ~Number~	~Postal~ ~City~
France	~Contact~	~Number~ ~Street~	~Postal~ ~City~
United States	~Contact~	~Number~ ~Street~	~City~ ~State~ ~Postal~

You would then join the record with the template table, and either allow a SQL stored procedure to format the address via the template or ask the application developers to write that code.

Government standards

Many government agencies have standard definitions already created, and typically you will be required to use these for any data exchange with a government entity. In this section, we will look at some example government standards across the globe.

United States

The U.S. government must love standards, because it writes so many of them. Here are a few you might need to use.

(FIPS) codes for states and counties

The FIPS (Federal Information Processing System) can be used to identify geographical regions of the country. There are state-level FIPS codes, which are two digits long; and there are county-level FIP codes, which are five digits long (the first two characters are the state FIPS code). If you need this level of addressing detail, you could include the FIPS code for a county, to help locate and group your tables by county.

The FIPS codes state/county list can be found here.

SIC codes

SIC (Standard Industry Classification) codes are used to categorize a business or entity by the type of work it does. There are four-digit numbers assigned by the U.S. government. If you are designing a database where the company business type is needed, the SIC code might a good standard to categorize your companies, particularly if only working with U.S. companies. SIC codes were first established in 1937 and were updated in 1987 as the economy changed. To this day, SIC codes are still being updated. It is still a very popular categorization system for businesses.

Find out more about SIC codes.

NAICS codes

The NAICS (North American Industry Classification System) codes are like SIC codes, but used by Canada, the U.S., and Mexico. NAICS codes are 6 digits long. The government stopped updating SIC codes in 1987 (although some private companies still do) in favor of the NAICS codes. The NAICS codes allow for a more precise business breakdown than the SIC code.

A NAICS code can be broken down to various levels of details. The top level (first two digits) represents the sector, such as information, construction, finance, and health care. Each subsequent digit further breaks down the category. The fifth digit provides the industry, and the sixth digit (final level) provides the U.S. industry. For example, the NAICS code of 511210 is for software publishers.

You can learn more about NAICS codes and the closely related NAPCS (North American Product Classification System) codes here.

If you are designing a new database that needs industry classification, consider using the NAICS codes. However, the SIC codes are still very common, so you might come across them in existing database systems.

European classification codes

NACE (Nomenclature of Economic Activities) is the European version of SIC/NAICS codes. The code begins with a section letter (B=fishing, D=manufacturing, and so on). The next two digits are a division (such as health), and the numbers after the decimal break it down further into group and class. For example, the code Q86.23 refers to dental practice.

Most of the countries in Europe and the United Kingdom use NACE codes for business classification. If you are designing databases for the European countries, you will likely need to work with NACE codes.

You can learn more about NACE codes here.

ISIC codes

SIC codes were originally developed by the United States, but in 1948, the United Nations created a similar set of codes called ISIC codes (International Standard Industrial Classification). The concept is like the United States SIC codes but used more often by member nations. The NACE code system is based on the ISIC codes.

You can learn about ISIC codes here.

You can convert codes online between the two systems using this website.

Whether you use NAICS codes or ISIC codes for business classification depends on your company list. With the United States only, NAISC codes are commonly recognized. If your market is international, you should consider ISIC or NACE codes instead.

Summary

In this chapter, we discussed internal standards and using user-defined data types to keep the database columns consistent. We also discussed a few examples of external standards. Depending on your application and industry, you should strive to use standards so that any data exchange with external companies are “speaking the same language.”

Chapter 7 Sample Data Models

In this chapter, we will look at some sample data models for common business applications. We will provide conceptual and logical model examples and leave the physical modeling up to you, depending on your database management system.

Human resources

Human resource (HR) departments are generally responsible for keeping track of employees and departments, their roles, managers, and so on. Some HR systems also track job applicants, and others include training requirements. Often, the HR department also handles payroll and benefits. Figure 25 shows a simple conceptual model for an HR department to track employees, departments, and applicants.

Conceptual model

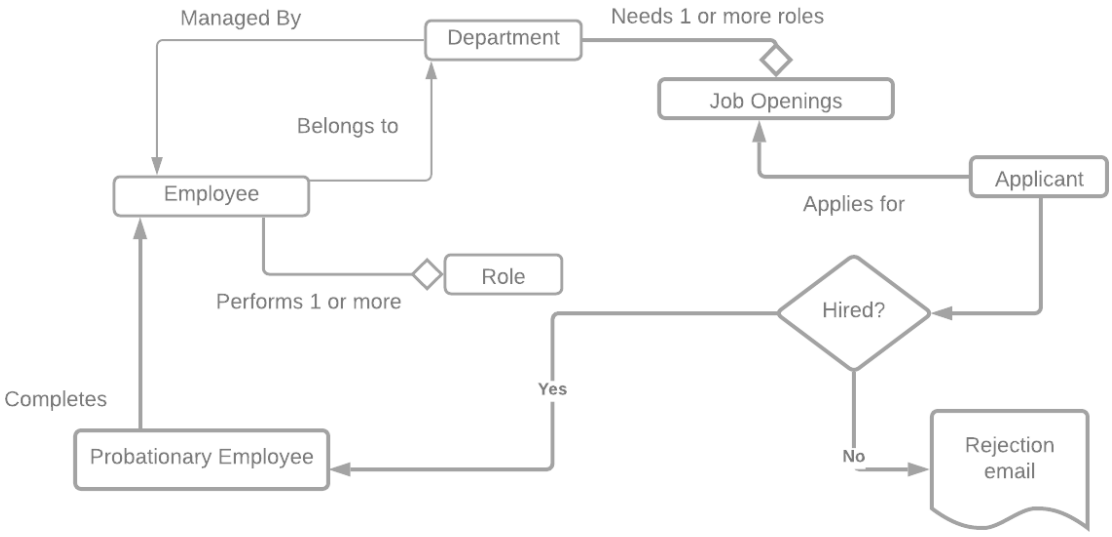


Figure 25: Human resources conceptual model

For this model, each employee belongs to one department, and each department is managed by one employee. The employee can perform multiple roles within the department. A department can have multiple job openings, which candidates can apply for. If the candidate is hired, they become a probationary employee, and after the probationary period, they become an employee. If the employee is not hired, a rejection email is sent.

While the model looks okay, after reviewing it with an HR employee, they asked what happens when an employee changes departments. This type of issue happens—something gets missed in the model. It is always a good practice to get as much review as possible of your conceptual model prior to starting the logical model. Based on this feedback from the user, we need to update the model some. Figure 26 shows the model update to the employee and departments.

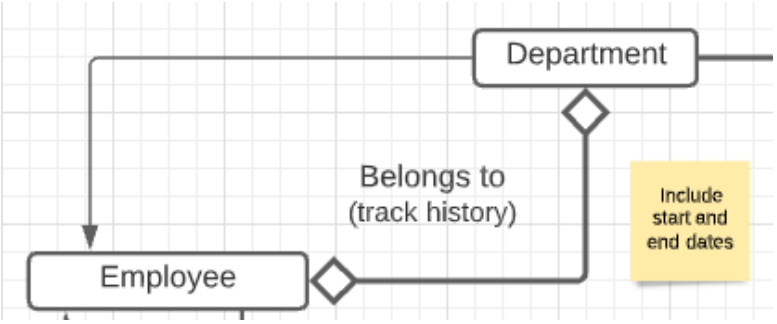


Figure 26: Update employee/department relationship

Logical model

Once you've got the approval for the conceptual model, you can begin to tackle the logical design phase. Figure 27 shows the logical model. Note that even after the logical model is designed, additional items (such as an applicant history table) might pop up.

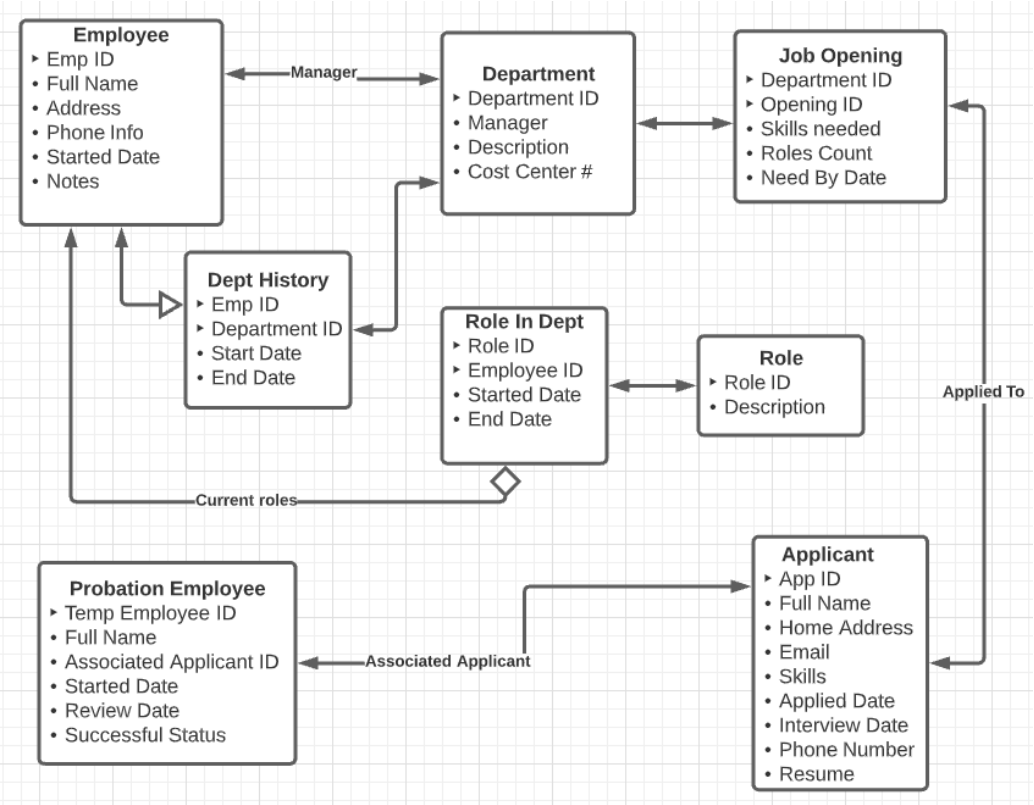


Figure 27: Human resource logical model

Accounting systems

A very common application for database systems is accounting. An accounting system consists of two key entities: accounts and transactions. The accounts list is referred to as the *chart of accounts*, while the transaction list is called the *ledger*.

Double entry accounting is a system where every transaction consists of debits and credits applied to different accounts. In each transaction, the total debits and credits must be equal. For some accounts, a debit represents an increase in the account balance, while in other accounts, the credit is the increase. Our database design must support debit/credit entries in a single transaction and a chart of accounts.

If you want to learn more about accounting systems, feel free to check out one of my previous books, *Accounting Succinctly*.

Conceptual model

The conceptual model for an accounting system is very simple, with only three primary entities, but frequently, a few associated entities (such as customers, vendors, and users). Figure 28 shows a sample accounting conceptual model.

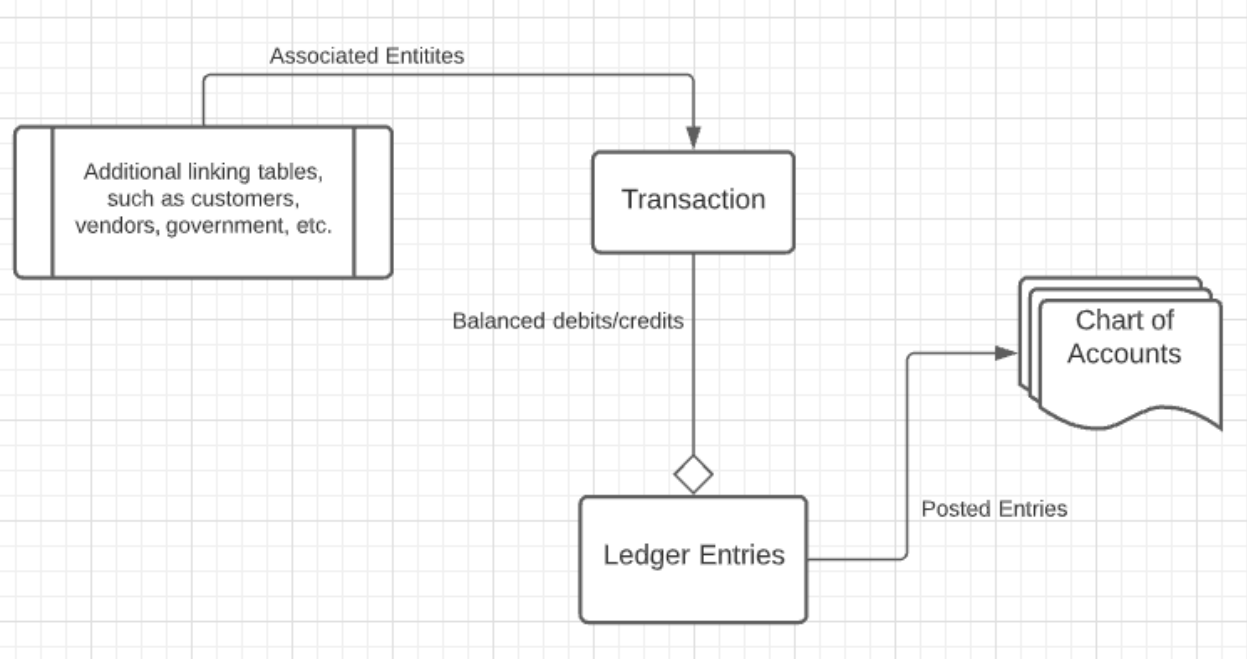


Figure 28: Accounting conceptual model

All standard financial reports will be generated from the chart of accounts. In addition, the transactions are typically placed in a journal, which specialized journals like cash receipts, cash disbursements, and a catch-all general journal. The transaction entity will likely have an attribute indicating the type of journal, although simple systems might place everything in the general journal.

Logical model

The logical model of the accounting entities is shown in Figure 29.

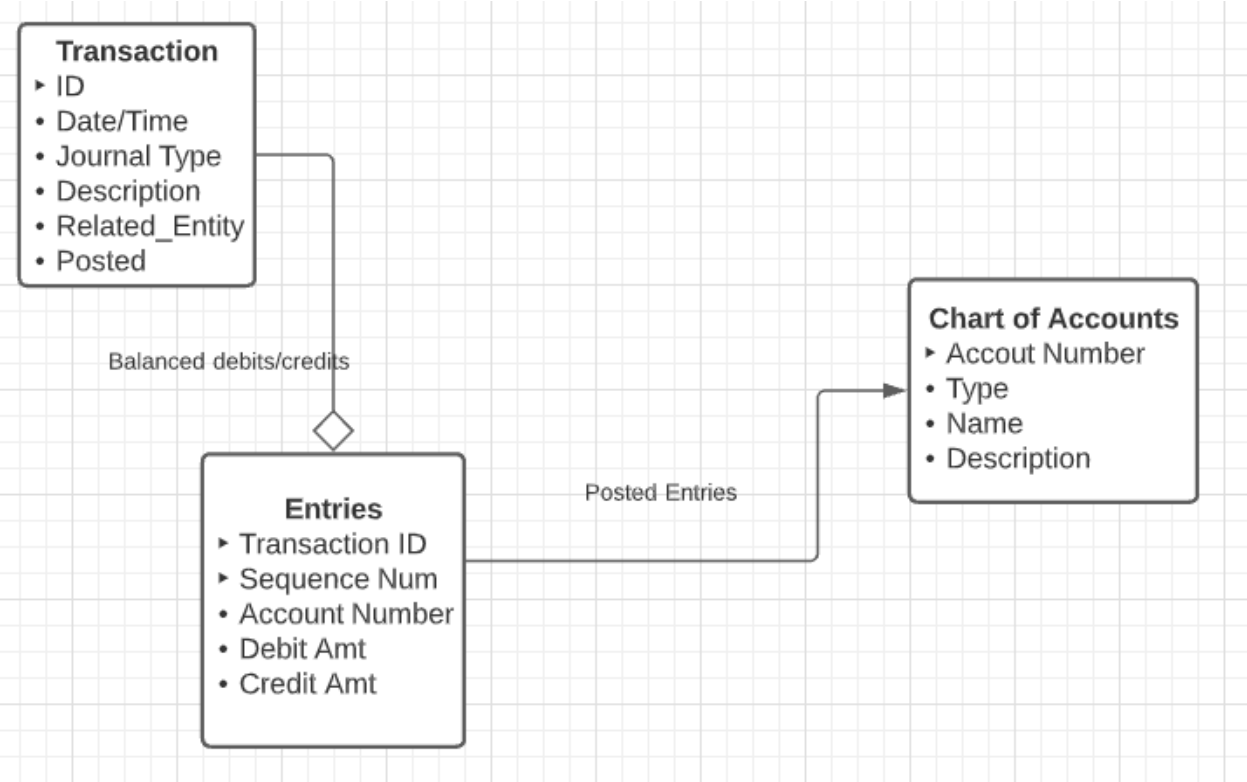


Figure 29: Logical accounting model

When a transaction is recorded, an ID and date and time are generated, and the user writes a description and possibly links the transaction to another entity. The transaction is the parent record to the child entries.

Each entry is associated with an account in the chart and will either debit or credit that account. It is a requirement of an accounting system that the total debit and credit entries for a transaction balance out.

For example, if a company were to buy a company car, they might create a transaction with a credit (reduces the balance) to the cash account and debit (increases the car inventory account) by the same amount.

While the business rules for transactions and accounting are complex, the data side is simple. The first book on double entry accounting was published in 1494 by Luca Pacioli, and for years was done manually.

Inventory

If a company sells or manufacturers a product, they likely will use a system to keep track of the inventory of product and/or materials. When a company only sells a product, they likely only need one entity to keep track of the inventory. However, if the company makes a product, they need different kinds of inventories for raw materials, works-in-progress, and finished goods.

Conceptual models

Figure 30 is a conceptual model for a manufacturing system.

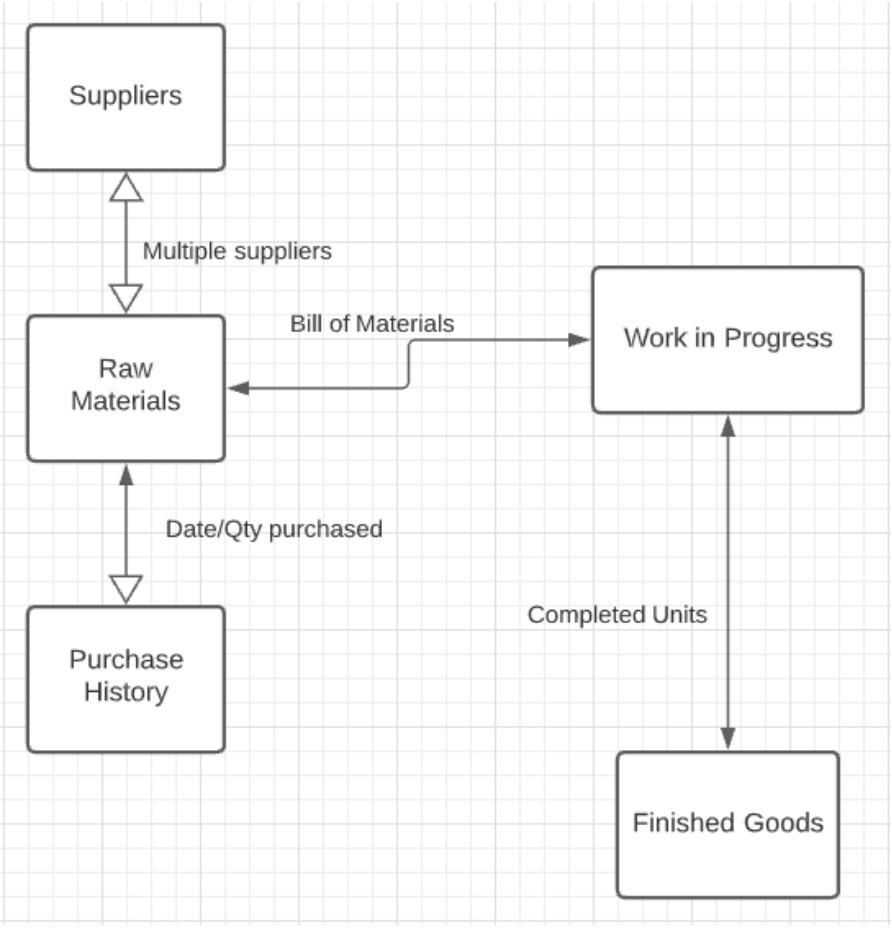


Figure 30: Conceptual manufacturing inventory

Material is purchased from suppliers and added to the raw materials needed to build the product. A bill of materials shows the raw inventory needed to create a product; while the work-in-progress inventory consists of the products that have been started, but not yet built. The finished goods inventory is the products that are ready to be sold to customers.

A very important consideration is that a date-sensitive purchase history is maintained, which we will discuss shortly.

Figure 31 shows an inventory conceptual model for a reseller (who purchases products and packages and ships them to customers).

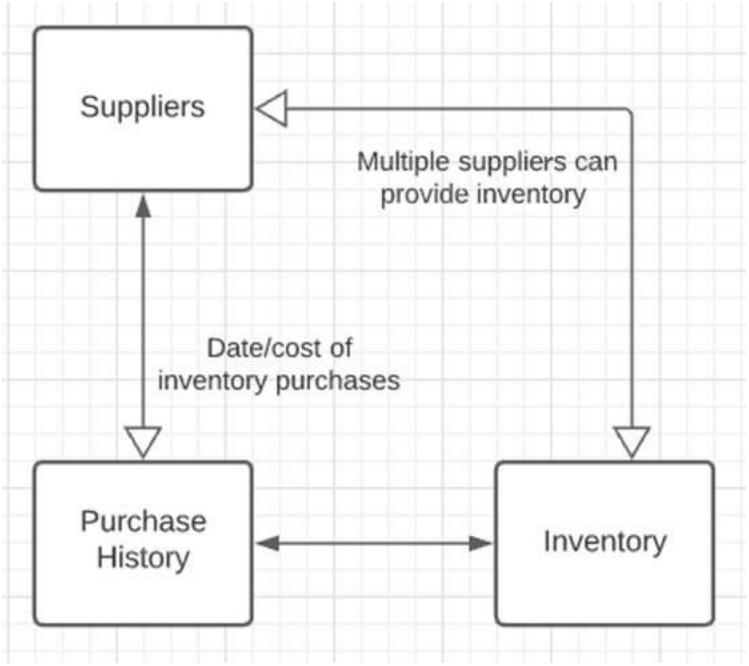


Figure 31: Reseller inventory model

Although there is only a single kind of inventory, the relationships between suppliers and inventory need to be tracked in a purchase history.

Cost of goods sold

In either inventory system, you must be able to determine how much an item in inventory costs (and hopefully was sold for more money). However, there are multiple approaches to determine the cost of the item.

Specific ID

In the case of rare or unique items (like cars or jewelry), each item has a serial number or some sort of identifier. The cost will be directly tied to this identifier.

LIFO (last in, first out)

In the LIFO costing method, the cost of your item is based on the last purchase price from your history. Since these prices are likely to be higher, this method reduces the amount of profit you show. While it is an allowed pricing method, it can also create some extra “profit” if your supply dwindles, and you sell some much older purchased items.

FIFO (first in, first out)

In this method, the first item purchased (the oldest one) is the first item sold. Assuming the first purchase prices are rising, this approach will show higher profit, since the earliest purchases likely have the lowest cost.

An example of the costing methods is shown in Table 21. In this table, we show how cost is computed when selling 32 products.

Table 21: Costing models

Date	Cost	Quantity	Using FIFO	Using LIFO
1/22/2019	$100	20	20*$100	None
3/4/2019	$125	15	12*$125	12*$125
7/5/2019	$150	20	None	20*$150
Cost			**$3,500**	**$4,500**

Note that the costing model chosen (in U.S. accounting rules) must be consistently applied and might not actually reflect the actual units sold. What is curious is that LIFO is allowed in the United States (which uses GAAP, generally accepted accounting principles), but banned in other accounting systems based on IFRS, International Financial Reporting Standards. This is one thing to be aware of when designing most accounting related systems; there are rules and regulations that must be followed but vary based on the accounting system standards used.

Logical model

The key takeaway from the conceptual model is the required purchase tracking entity, regardless of whether purchasing raw materials or reselling items purchased from a vendor. Figure 32 shows a simplified logical mode of the key entities.

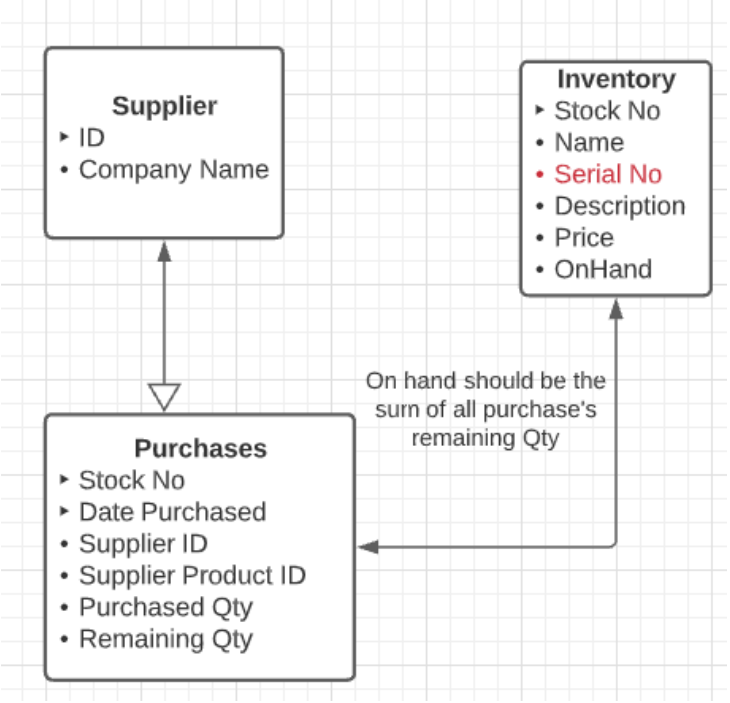


Figure 32: Inventory key entities

You may need additional fields, perhaps images of the inventory item, but it is most important to track the purchase history. The serial number attribute might not be required, unless the product carries some sort of identifier, such as a car's VIN.

Customer orders

A very common business application is tracking customers and ordered products. Frequently, a component of customer orders is the inventory model we discussed previously. Order items will reference an item number from the inventory (regular inventory or finished goods).

Conceptual model

Figure 33 shows a conceptual model for the order and invoicing system.

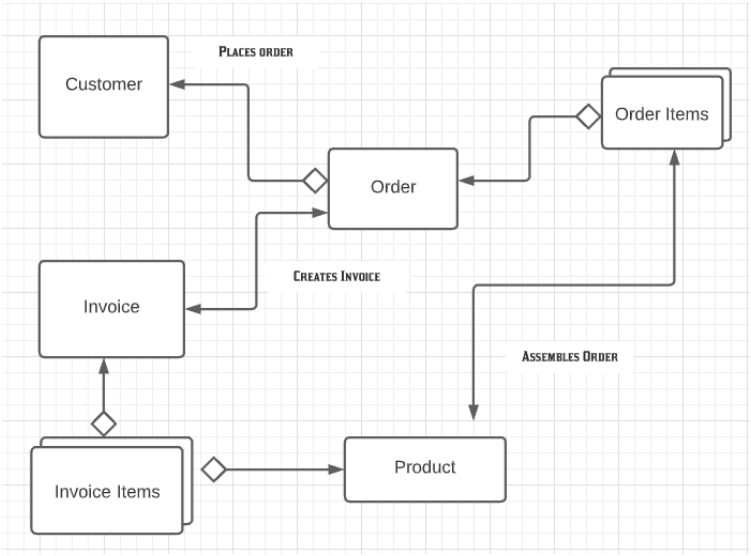


Figure 33: Customer orders

A customer places (hopefully many) orders with the company. Each order has a collection of items (pulled from inventory). The order also produces an invoice, which has a list of those same items, but with pricing information.

Logical model

The logical model is shown in Figure 34, adding attributes to the various entities.

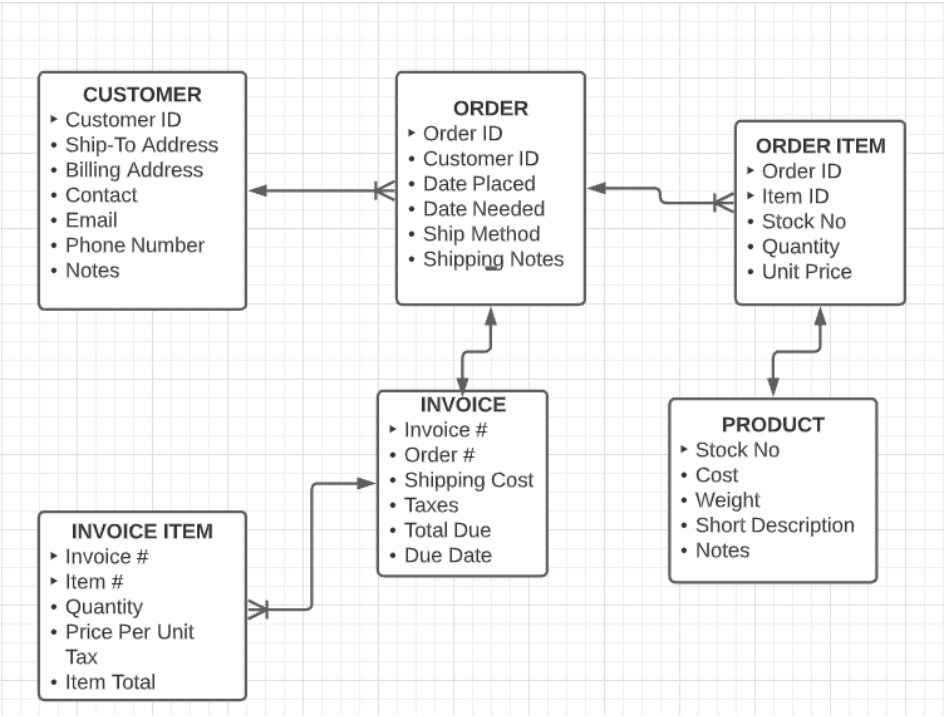


Figure 34: Customer order logical model

Reservations

A reservation system is a good type of application for computerization. The system tracks resources (hotel rooms, sports fields, conference rooms) for time periods. It should be able to indicate who is using a particular resource at a given time and to search for time slots where the resource is available.

Note that the entity types might change (hotel, sports club) as well as the resource (rooms, courts)—but not the concept, which is a group of resources available with a facility that needs to be reserved and searched for availability.

Conceptual model

Figure 35 shows the conceptual model for a general reservation system. The facility and resource entities will likely be named differently depending on the type of facility (hotel, sports cub, car rental, and so on).

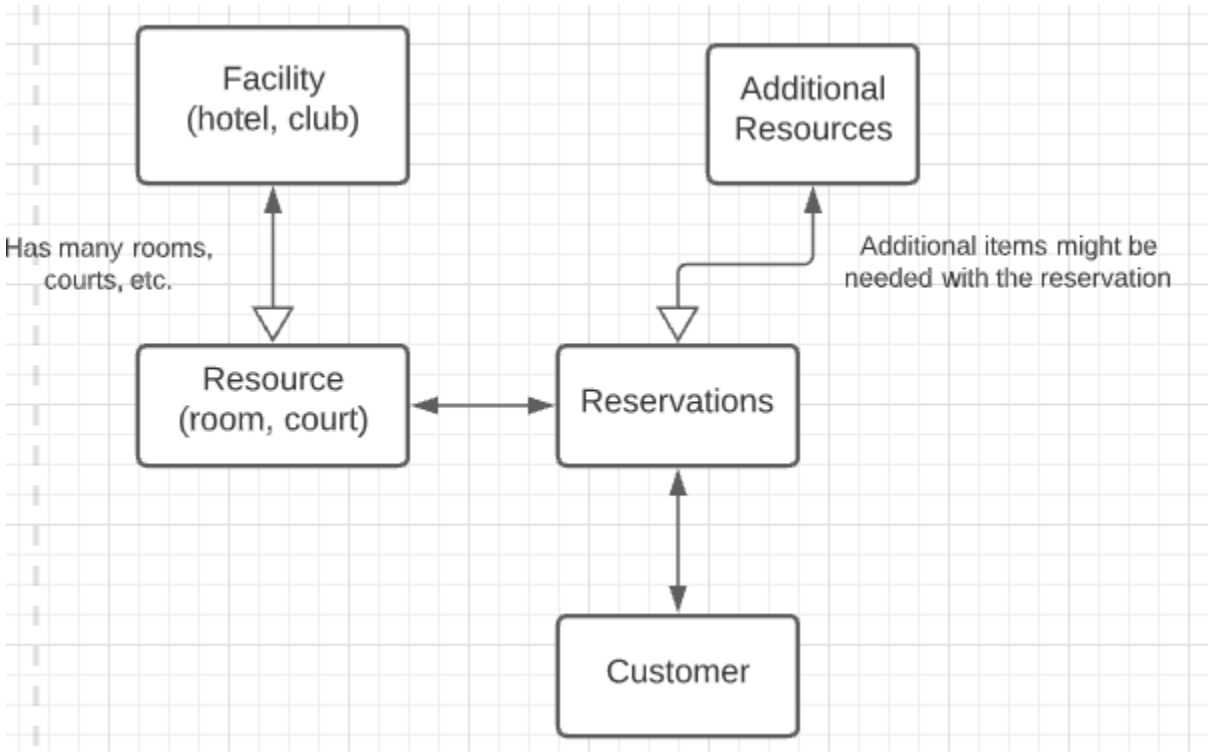


Figure 35: Reservation conceptual model

Logical model

The logical model will provide details to the various entities. The customer, resource name, etc., is likely to stay the same. The key differences are that the reservation length of time might vary (tennis courts rented by the hour, hotel rooms by the day) and the number of customers.

Figure 36 shows the reservation system logical model.

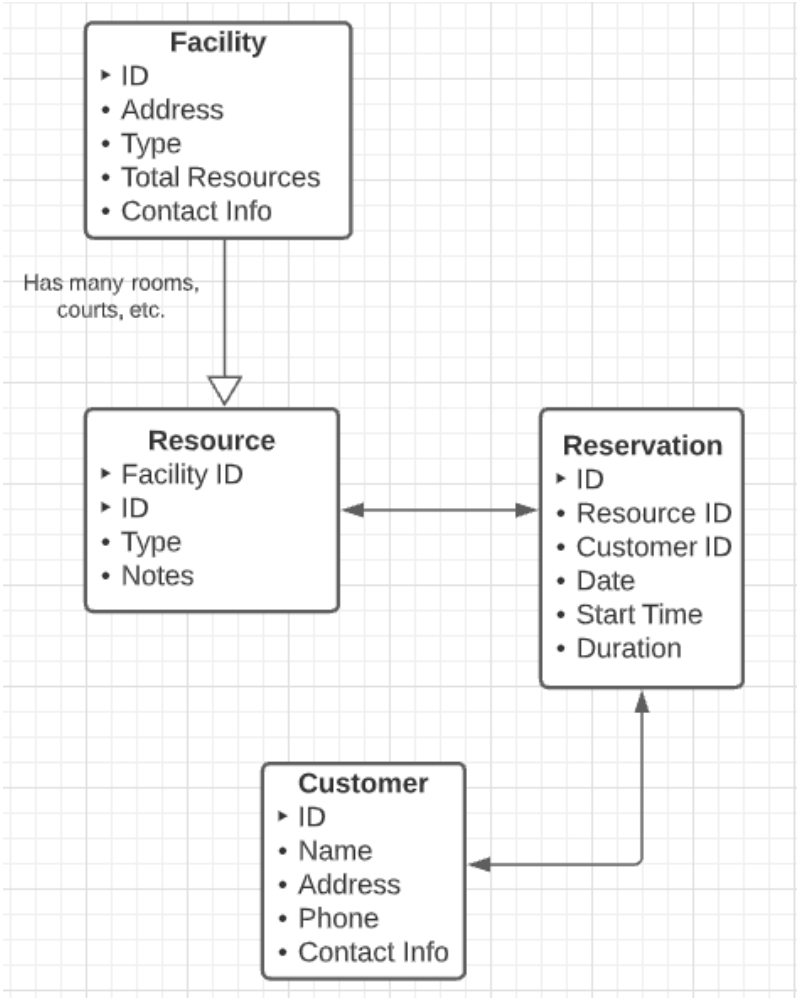


Figure 36: Reservation system logical model

The reservation entity will provide enough details, like date and customer, for the reservation. Searching for an open slot would require a date range and other parameters (such as length of stay and size of resources). It should then list open slots, depending on the search parameters.

Tip: If you want to annoy the person searching, don't show all open slots, but make them click on the date, only to tell them there is nothing available. Sadly, many reservation systems do just that.

Sports

This conceptual and logical model can be used as the basis for a sports league (soccer, baseball, and so on). It tracks the teams and players, schedule, and results. It also tracks referees associated with the scheduled games and locations (fields, stadiums) where the games will be played.

Conceptual model

Figure 37 is the conceptual model. Each team has any number of players. Two teams will meet for each game, and one or more referees. We also link the playing field and the player from the team that played in the game.

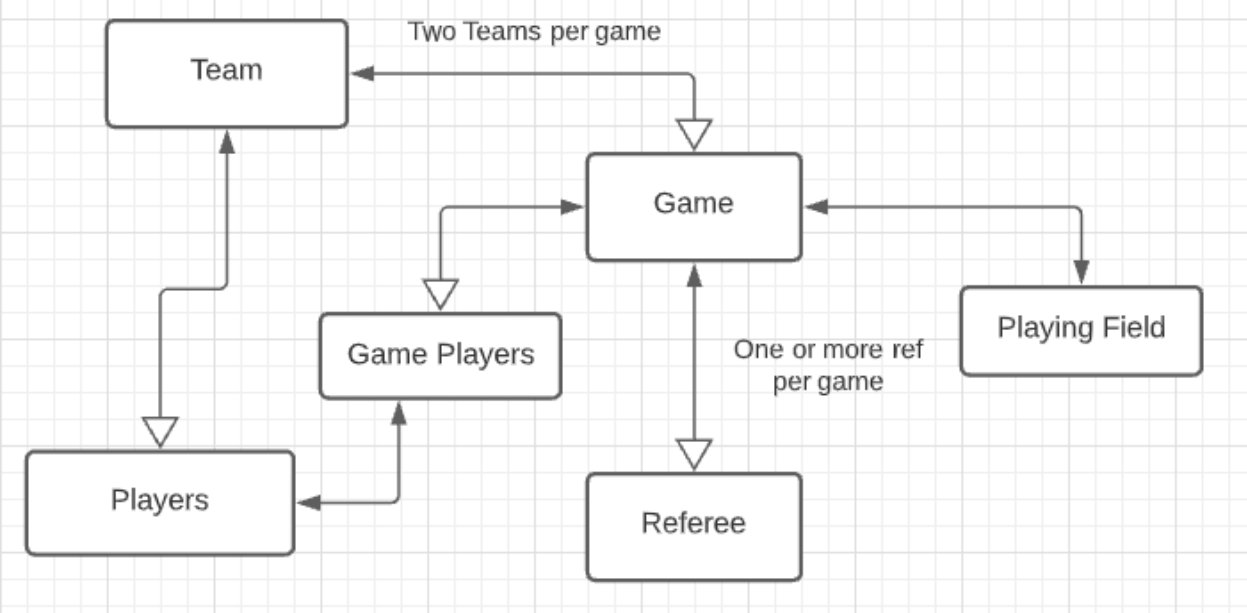


Figure 37: Sports conceptual model

Logical model

Figure 38 shows the logical model, which only tracks game wins and losses. Depending on the sport, you might want to add additional player level statistics, such as goals scored or runs allowed.

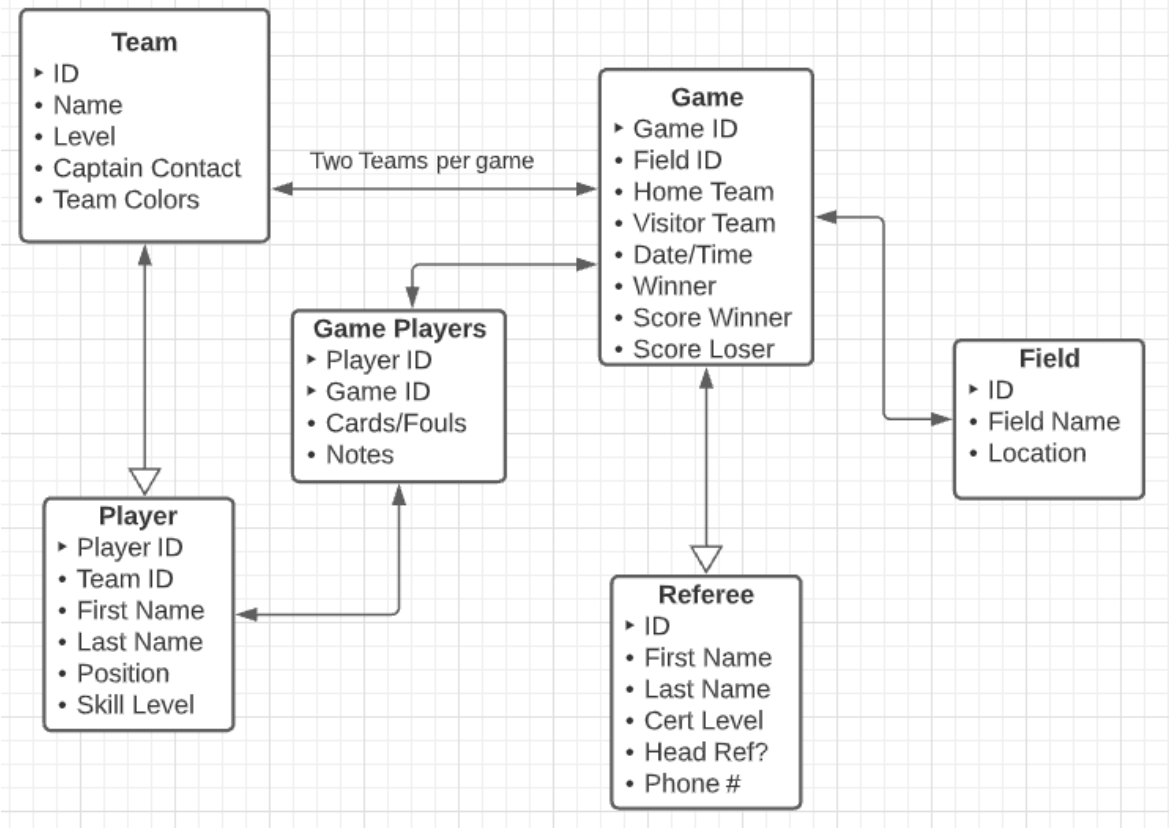


Figure 38: Sports logical model

The model might be adjusted based on the sport. For soccer, you might want to track red cards issued to a player. For American football, players can be ejected from the game.

You can also use the Game entity to determine standings. You could assign points per win or tie—the highest number of points would be first place in standings. Things get a bit more complex when ties occur, and you need to compute tiebreaker rules. The game table provides the information you'll need to report standings.

Summary

In this chapter, we looked at some types of applications and the conceptual and logical models for them. These models are a starting point to give you some ideas. However, be sure to create your models after talking things over and hunting down requirements with the users. In the next chapter, we will take the logical model from the accounting application and create a physical model for it in Microsoft SQL Server.

Chapter 8 Physical Model

In this chapter, we are going to take the logical accounting model and convert it to a physical model using Microsoft SQL Server. We will use constraints and foreign keys, and well as attribute descriptions, to create a well-designed and documented database system.

Figure 39 is the logical model we designed in Chapter 7.

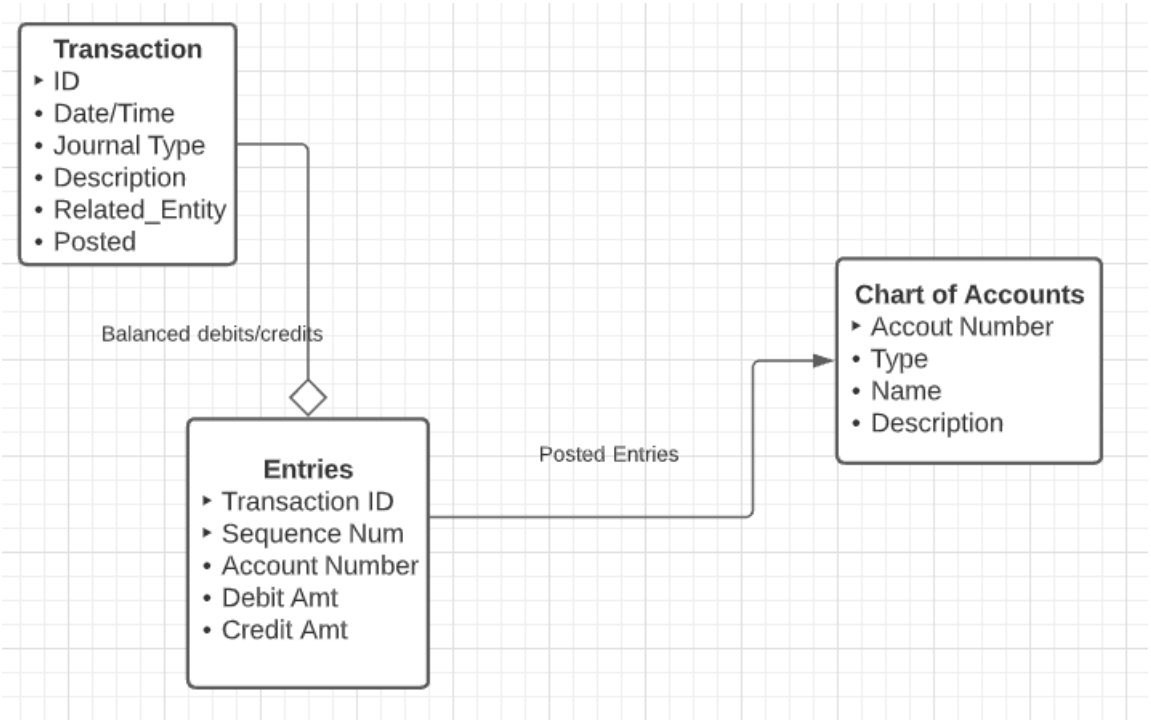


Figure 39: Accounting system logical model

Lookup tables

There are two lookup tables we can create: one for the journal type, and the second for the account type in the chart of accounts. Often the lookup tables might not be on the logical design, but their need will become evident as the physical model is built.

Journal type

The journal type lists the types of specialized journals the business wished to keep. Journal entries themselves are transaction records; the type is used to group related transactions together, based on the type of transaction.

Code Listing 3 shows the code to create the lookup table and populate it with some common journal types.

Code Listing 3: Journal type

```
create table dbo.JournalType
(
      ID char(2) PRIMARY KEY,
      ShortName varchar(32) NOT NULL UNIQUE
)
INSERT INTO dbo.JournalType (ID,ShortName)
values
('GJ','General Journal'),
('CR','Cash Receipts'),
('CD','Cash Disbursements'),
('SJ','Sales Journal'),
('PJ','Purchases Journal')
```

Depending on the system design, you might want to add additional fields, possibly a list of required accounts (for example, the CR journal requires a debit to cash, or SJ must have a sales account). Generally, the specialized journals will always have certain accounts, where the general journal is the catchall for other entries.

Account type

There are five basic account types in the chart of accounts. These are shown in Table 22.

Table 22: Basic account types

Type	Debit	Description
ASSETS	Increases	Things a business owns: cash, inventory, etc.
LIABILITIES	Decreases	Debts of the business.
EQUITY	Decreases	Portion of business assets owned by owners or shareholders.
REVENUE	Decreases	Money brought in by the business.
EXPENSES	Increases	Money spent to keep business running.

There can be other types, but for most systems, those five are the parent types. Often, the account numbering scheme can be used to determine the account type. For example, if we use cash (an asset) to pay the phone bill (expense), we might create a transaction in the cash disbursement journal with the following entries, shown in Table 23.

Table 23: Transaction entries to pay phone bill

Account name	Debit	Credit	Notes
PHONE_BILL	$500		Record expense to pay phone bill.
CASH-Checking		$500	Decrease cash in checking account.

Our code to create and populate the account type table is shown in Code Listing 4.

Code Listing 4: Account types

```
create table dbo.AccountType
(
      ID        varchar(12) PRIMARY KEY,
      ShortName varchar(32) NOT NULL UNIQUE,
      Category  varchar(12)
              CHECK( Category in
              ('ASSET','LIABILITY','EQUITY','REVENUE','EXPENSES'))
                NOT NULL,
      ListOrder int NOT NULL
)
INSERT INTO dbo.AccountType(ID,ShortName,Category,ListOrder)
values
('CURASSET','Current Assets','ASSET',110),
('LTASSET','Long Term Assets','ASSET',120),
('CURLIAB','Current Liabilities','LIABILITY',210),
('LTLIAB','Long Term Liabilities','LIABILITY',220),
('OWNER','Owner Equity','EQUITY',310),
('RETAINED','Retained Earnings','EQUITY',320),
('REVENUE','Sales','REVENUE',410),
('INTEREST','Earned Interest','REVENUE',410),
('EXPENSES','Current Expenses','EXPENSES',510)
```

With these two lookup tables, we can build out our base tables per the logical design.

Chart of accounts

The chart of accounts is the primary reporting source for an accounting system. In very simple terms, a balance sheet is a list of the assets, liabilities, and equity accounts. An income statement is a list of the revenue and expense accounts. (There is more detail needed for each report, but the source of the reports is the chart of accounts.)

Code Listing 5 shows the code and some accounts for the chart.

Code Listing 5: Chart of accounts creation

```
create table dbo.Chart_of_Accounts
```

```
(
      Account_Number varchar(8) PRIMARY KEY,
      AcctType varchar(12) FOREIGN KEY REFERENCES AccountType(ID) NOT
NULL,
      AcctName varchar(50) NOT NULL,
      Balance money default 0 NOT NULL,
      [Description] varchar(max)
)
INSERT INTO Chart_of_Accounts (Account_Number,AcctType,AcctName)
VALUES
('1000','CURASSET','Cash-Checking'),
('1010','CURASSET','Cash-Savings'),
('1500','LTASSET','Company Truck'),
('1510','LTASSET','Forklift'),
('1520','LTASSET','Shipping Equipment'),
('1600','LTASSET','Warehouse/Office'),
('2000','CURLIAB','Vendor Payables'),
('2010','CURLIAB','Payroll Taxes'),
('2500','LTLIAB','Building Mortgage'),
('3000','OWNER','Owner Equity')
```

If we were to view the chart of accounts initially, all the accounts would have a zero balance. Table 24 shows the initial balance sheet.

Table 24: Initial balance sheet

Account_Number	AcctType	AcctName	Balance
1000	CURASSET	Cash-Checking	0.00
1010	CURASSET	Cash-Savings	0.00
1500	LTASSET	Company Truck	0.00
1510	LTASSET	Forklift	0.00
1520	LTASSET	Shipping Equipment	0.00
1600	LTASSET	Warehouse/Office	0.00
2000	CURLIAB	Vendor Payables	0.00
2010	CURLIAB	Payroll Taxes	0.00
2500	LTLIAB	Building Mortgage	0.00
3000	OWNER	Owner Equity	0.00

The account number is assigned, and the common approach is to use numbers in the thousands to represent the different account types. This numbering system is simple, typically for a small business. Larger businesses often break down the chart into much more detail, and the account numbers likely reflect cost centers, further breakdown of payroll taxes, and so on.

Transaction

The transaction table is a header record describing the entries that apply to the chart. Code Listing 6 shows the code to create the transaction table.

Code Listing 6: Create transaction table

```
create table dbo.[transactionDetail]
(
      ID           INT  identity(1,1) PRIMARY KEY,
      TransDate    DATETIME DEFAULT getDate() NOT NULL,
      JournalType  CHAR(2)
          DEFAULT 'GJ' FOREIGN KEY
                       REFERENCES JournalType(ID) NOT NULL,
      ShortDesc        varchar(100) NOT NULL,
      TransactionNotes varchar(max),
      Posted           tinyint Default 0 NOT NULL,
      CONSTRAINT Check_Posted CHECK ( Posted=0 or
dbo.SumTransactions(ID)=0)
)
```

Note that we called the table **`TransactionDetail`**. **Transaction** would be a wonderful name, except that **Transaction** is a reserved word in almost all SQL database management systems. The **`Posted`** flag indicates whether this particular transaction has been applied to the chart of accounts. It defaults to zero (not yet posted) and has an interesting **`CHECK`** option.

`CHECK` constraints are usually based on columns within the current table. However, for a transaction to be valid (so it can be posted), it must check the linked Entries table. While this cannot be directly done, SQL Server allows you to create a SQL user-defined function that will perform the test for us. Code Listing 7 shows the code.

Code Listing 7: SumTransactions function

```
CREATE function dbo.SumTransactions(@ID int)
RETURNS money
AS
BEGIN
      DECLARE @ans Money
      SELECT @ans =Sum(debitAmt)-Sum(CreditAmt)
              FROM Dbo.EntryDetail WHERE TransactionID=@id
      RETURN @ans
END
```

By moving the code to a UDF (user-defined function), we can work around the **CHECK** limitation to columns within the table. Other DBMS products might handle the situation differently, so you need to be comfortable with the nuances of your SQL database. If the **Posted** flag is 0, we don't check the condition of balanced debits and credits. When we attempt to set the flag, the **SumTransaction** function gets called and will only return **TRUE** to the constraint if the linked entries balance out.

Entry

The **EntryDetail** table contains the accounts and debit/credit amounts that form the transaction. Code Listing 8 shows the code to create the **EntryDetail** table.

Code Listing 8: EntryDetail table

```
create table dbo.EntryDetail
(
      TransactionID     INT FOREIGN KEY REFERENCES
[TransactionDetails](ID),
      SequenceNumber    INT NOT NULL,
      Account_Number    VARCHAR(8) NOT NULL
                        FOREIGN KEY
REFERENCES  [Chart_of_Accounts](Account_number),
      DebitAmt          SmallMoney DEFAULT 0,
      CreditAmt         SmallMoney DEFAULT 0,
      PRIMARY KEY (TransactionID,SequenceNumber),
      -- Require either a debit or credit, but not both
      CONSTRAINT Entry_check
      CHECK ( (DebitAmt<> 0 OR CreditAmt <> 0)
             AND (DebitAmt * CreditAmt = 0) )
)
```

This table collects all the chart of account entries needed to complete the transaction. It also prevents a user from having both the debit and credit amounts supplied and from both amounts being zero.

This constraint is important since the SQL user-defined function from earlier relies on these conditions.

Putting it together

Now, with all our tables created and populated, we need to open the business with the first transaction. The business owner is going to deposit $100,000 USD into the business, splitting it between the checking and savings accounts.

Our first transaction is shown in Code Listing 9.

Code Listing 9: Post initial deposit to business

```
DECLARE @transID int
INSERT INTO transactionDetail
(JournalType,ShortDesc,TransactionNotes,Posted)
VALUES ('GJ','Owner added money to start business',
       'The sole owner of the business deposited $100,000 '+
       'in cash to start the business',0)
SET @transID = IDENT_CURRENT('transactionDetail')

INSERT INTO EntryDetail VALUES
(@transID,1,'1000',10000,0),
(@transID,2,'1010',96000,0),
(@transID,3,'3000',0,100000)

UPDATE transactionDetail set posted=1 where ID = @transID
```

The **IDENT_CURRENT** function is a Microsoft SQL Server function that gets the most recent identity key from the specified table.

If we run this code, we will get an error trying to set the posted flag to 1.

The **UPDATE** statement conflicted with the **CHECK** constraint **"Check_Posted"**.

The issue is that the debit total is $106,000 and the credit total is $100,000. The second entry is an error; it should be $90,000. So, we fix that, and try again. This time, the **Posted** flag can be set to 1 since the debits and credits are in balance. Once the transaction is posted, our balance sheet from the chart of accounts looks like Table 25.

Table 25: Balance sheet view after first posted transaction

Account	AcctType	AcctName	Debit Bal	Credit Bal
1000	CURASSET	Cash-Checking	$10,000.00	
1010	CURASSET	Cash-Savings	$90,000.00	
1500	LTASSET	Company Truck	$0.00	
1510	LTASSET	Forklift	$0.00	
1520	LTASSET	Shipping Equipment	$0.00	
1600	LTASSET	Warehouse/Office	$0.00	
2000	CURLIAB	Vendor Payables		$0.00
2010	CURLIAB	Payroll Taxes		$0.00
2500	LTLIAB	Building Mortgage		$0.00

Account	AcctType	AcctName	Debit Bal	Credit Bal
3000	OWNER	Owner Equity		$100,000.00

So, the owner equity account is $100,000 (their deposit) and the cash is split between checking and savings.

Summary

While the table structure of an accounting system is simple, you can see how we need to add constraints when converting our logical model to a physical model to be represented in the database system. Our normalized accounting model is robust enough to handle the rules of double entry accounting and provide a solid basis for the company's financial reporting needs.

Chapter 9 Summary

Users often have a good understanding of their data needs but are not sure how to express it or put it into a database. Users are concerned with running their business, while developers are concerned with creating a robust database system. In this book, we covered the steps necessary to get from user information into a database system.

Each database management system has its nuances, and while SQL itself is a general language, each vendor will likely have features unique to their product. If you are working in a single DBMS, you can apply these nonstandard vendor features and improve your design. If you need to build a system that works across different database product, make sure your basic design of tables and views use standard SQL.

It seems like a lot of work to design various models, normalize database schemas, figure out view and indexes, and so on. However, a robust database design will save time in the long run, and likely it will prevent those emergency requests, like if a product is shipped to the wrong address, or a client needs to add more children to the family record, for example.

Be sure to talk to the right people and get the conceptual model very well defined before you start the logical model. Define the logical normalized model before you create the database (physical model)—you'll be glad you did!

Appendix A DBMS Field Types

The physical model will be created using the constructs provided by the chosen database management tool. In Chapter 5, we highlighted the table names, field types, and so on. This appendix lists these various constructs in different DBMS solutions.

Table 26 shows the various string data types between SQL database management systems.

Table 26: DBMS string data types

Description	MS-SQL Server	mySQL	Oracle
Fixed size	`CHAR`	`CHAR`	`CHAR`
Varying size	`VARCHAR`	`VARCHAR`	`VARCHAR2`
Unicode fixed size	`NCHAR`	See (a)	`NCHAR`
Unicode varying size	`NVARCHAR`	See (a)	`NVARCHAR2`
Unlimited text	`[n]VARCHAR(max)`	`TEXT`	`[n]CLOB`
Enumerated values	n/a	`ENUM`	n/a
Array/set of values	n/a	`SET`	n/a
Binary data	`BINARY, VARBINARY`	`BINARY, VARBINARY`	`CLOB`
Blob (large binary object)	`BINARY`	`BLOB, TEXT`	`BLOB`
Text	*Deprecated*	`TEXT`	
Image	`IMAGE`	`BLOB`	`RAW`
GUID value	`UNIQUEIDENTIFIER`	`BINARY(16)`	`RAW sys_guid()`

(a) Unicode support is provided by attaching the appropriate collation to the **`CHAR`** or **`VARCHAR`** definition.

String data types can be used for any kind of data, and often you will see systems where the character field contains numeric or date values. More often, these values cause problems down the road, so try not to rely overly on character data when other types are a better fit.

If you do use character fields, consider constraining the allowed values. Use foreign keys or check constraints to limit the type of data in the field. For example, if you paste content from Microsoft Word into a field, you could end up with different characters than expected.

Table 27 shows the various numeric data types between SQL database management systems.

Table 27: DBMS numeric types

Description	MS-SQL Server	mySQL	Oracle
Boolean (bit)	`BIT`	`BIT`	`NUMBER(3)`
Tiny Int *0-255*	`TINYINT`	`TINYINT`	`NUMBER(3)`
Short Integer *-32,768 and 32,767*	`SMALLINT`	`SMALLINT`	`NUMBER(5)`
Medium Integer -8,388,608 and 8,388,607	*n/a*	`MEDIUMINT`	*n/a*
Integer *-2,147,483,648 and 2,147,483,647*	`INT`	`INTEGER`	`NUMBER(10)`
Long Integer -2^{63} TO 2^{63}	`BIGINT`	`BIGINT`	`NUMBER(19)`
Decimal	`DECIMAL`	`DECIMAL`	`NUMBER`
Numeric	`NUMERIC`	`NUMERIC`	`NUMBER`
Float	`FLOAT`	`FLOAT`	`BINARY_FLOAT`
Real	`REAL`	`DOUBLE`	`BINARY_DOUBLE`
Money	`SMALLMONEY`, `MONEY`	*Use Numeric*	`NUMBER(10,4)`

Oracle stores all numeric values as **`NUMBER`** and has deprecated the integer data type. You can use the number data type with the specified precision to store integer values.

By its size, a numeric data type can constrain the possible values. However, it is generally a good idea to constrain it via a check constraint. Storing a birth year as an integer, when dealing with living people, should probably not allow dates prior to 1900 or after the current year.

Note: If any of my readers were born prior to 1900, please accept my apologies.

Table 28 lists the various date data types. While it should go without saying to use a date type whenever possible, you still will likely encounter systems that use character fields to hold date values. I strongly recommend avoiding that approach, as you'll save yourself quite a bit of aggravation down the road.

Table 28: Date data types

Description	MS-SQL Server	mySQL	Oracle
Date *1/1/0001 to 12/31/9999*	DATE	DATE	DATE
Date with Time 1/1/1753 to 12/31/9999 00:00:00 to 23:59:59:997	DateTime	DATETIME (Time only to 23:59:59)	DATE *(fractional seconds are truncated)*
Date with Time *1/1/0001 to 12/31/9999* *00:00:00 to 23:59:59:*9999999	DateTime2	*n/a*	n/a
Date time with GMT offset *1/1/0001 to 12/31/9999* *00:00:00 to 23:59:59:* 9999999 *Offset:* -14:00 to 14:00	DateTimeOffset	TimeStamp	RAW
Small date time 1/1/1900 to 6/6/2079 00:00:00 to 23:59:29	SmallDateTime	n/a	n/a
Time *00:00:00 to 23:59:59:*9999999	Time	TIME *(-839:59:59 to 838:59:59) since can also be a time range*	TIME