DATA MODELING FOR MongoDB

Building Well-Designed and Supportable MongoDB Databases

STEVE HOBERMAN

Foreword by Nike Information Architect Ryan Smith

Building Well-Designed and Supportable MongoDB Databases

first edition

Steve Hoberman

Published by:

Technics Publications, LLC

2 Lindsley Road

Basking Ridge, NJ 07920 USA

http://www.TechnicsPub.com

Cover design by Mark Brye

Edited by Carol Lehn and Erin Elizabeth Long

Technical reviews by Rob Garrison and Richard Kreuter

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without written permission from the publisher, except for the inclusion of brief quotations in a review.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

MongoDB® is a registered trademark of MongoDB, Inc. All other trademarks are property of their respective owners and should be treated as such.

Copyright © 2014 by Technics Publications, LLC

ISBN, print ed. 978-1-935504-70-2

ISBN, ePub ed. 978-1-935504-71-9

First Printing 2014

Library of Congress Control Number: 2014938921

To my brother, Gary, who is not only an impressive techno-wiz and CIO, but also knows how to apply technology to make amazing things happen.

Table of Contents

Foreword

By Ryan Smith, Information Architect at Nike

How do you design for a database platform which doesn't require design? Should you design your data at all, or just start building?

I'd like to think I have a somewhat unique perspective on this, working at Nike where I have a front row seat to world class product design and innovation.

Perspectives vary on the importance of designing data these days. Traditional databases fundamentally force a certain level of design, to define the structure of your information before content can be inserted. But many NoSQL databases will accept whatever data format you throw at them, and you can completely change that format from one record to the next. So in theory you don't need to have a plan for your data at all!

I strongly believe that you need to have a plan. Designing your data brings clarity. It exposes what you do and don't know about it, and what confusion may exist about it within your team and organization. Designing data is an essential form of leadership, charting a course and testing ideas, bringing forward a clear story while being open to changing it based on new learning.

Most importantly, design is about understanding the consumer, knowing what audiences will be using the system and what they need the data to do for them. Data modeling can greatly accelerate development, improve the quality of the end product, reduce maintenance costs, and multiply the value of what has been built by establishing a common understanding of the information.

In many ways up-front data design with NoSQL databases can actually be *more* important than it is with traditional relational databases. For example, since many of these databases don't join data sets, their fast-retrieval performance potential excels only if the collections you've created contain all or most of the information required

by a given request. If you have to retrieve a large percentage of your database content in order to scan and filter it down to a relevant result set, you would have been better off writing SQL to join tables within a relational database. But as I have seen in my experience, good data design practices can lead to excellent performance.

Beyond the performance topic, NoSQL databases with flexible schema capabilities similarly require *more* discipline in aligning to a common information model. For example, MongoDB would not hesitate to allow you to sequentially save four records in the same collection with field names of: **zipCode**, **zipcode**, **ZipCode**, and **postalCode**, respectively. Each of these variations will be treated as a new field, with no warnings given. Everything will work great until you ask for your **zipCode** values back and only one document out of four has a field by that name.

The flexible schema is a great innovation for quick evolution of your data model, and yet it requires discipline to harvest the benefits without experiencing major data quality issues and other frustrations as a result.

Data modeling is essential for success, but it's not rocket science, and with this book it is easier than ever to implement effectively. Written by an exceptional teacher of data modeling, these chapters clearly explain the process needed to navigate these new capabilities with confidence. Anyone who has taken Steve's Data Modeling Master Class can attest to his passion for both data and teaching. In reading the manuscript, it was immediately evident to me that his gift for teaching is here in his writing every bit as much as when he presents in person. Steve carefully crafts his explanations so that even the more abstract concepts can be easily grasped and internalized. Steve's broad consulting experience also shows through.

Seth Godin states in his book Linchpin that there is a "thrashing" process inherent in the creation of any product, meaning the brainstorming and iteration of different ideas and approaches. He writes, "Thrashing is essential. The question is: when to thrash? In the typical amateur project, all the thrashing is near the end…Professional creators thrash early." Anyone who has found themselves at the tail end of a poorly-executed project knows all about late thrashing. Early and ongoing modeling adds tremendous value, no matter how agile your methodology.

Steve's book is about how to work through uncertainties like a professional, so you can evolve your data models over time without devolving into chaos.

May your data models provide calm and clarity which allow your work to thrive, so that you can focus on what matters most: your team, your objectives, and your consumers.

Introduction

Congratulations! You completed the MongoDB application within the given tight timeframe, and there is a party to celebrate your application's release into production. Although people are congratulating you at the celebration, you are feeling some uneasiness inside. To complete the project on time required making a lot of assumptions about the data such as what terms meant and how calculations were derived. In addition, the poor documentation about the application will be of limited use to the support team, and not investigating all of the inherent rules in the data may eventually lead to poorly performing structures in the not-so-distant future.

Now, what if you had a time machine and could go back and read this book before starting the project. You would learn that even NoSQL databases like MongoDB require some level of data modeling. **Data modeling is the process of learning about the data, and regardless of technology, this process must be performed for a successful application.** You would learn the value of conceptual, logical, and physical data modeling and how each stage increases our knowledge of the data and reduces assumptions and poor design decisions.

Read this book to learn how to do data modeling for MongoDB applications and accomplish these five objectives:

1. Understand how data modeling contributes to the process of learning about the data and is, therefore, a required technique even when the resulting database is not relational. That is, *NoSQL* does not mean *NoDataModeling*!
2. Know how NoSQL databases differ from traditional relational databases and where MongoDB fits.

3. Explore each MongoDB object and comprehend how each compares to its data modeling and traditional relational database counterparts, as well as learn the basics of adding, querying, updating, and deleting data in MongoDB.
4. Practice a streamlined, template-driven approach to performing conceptual, logical, and physical data modeling. Recognize that *data modeling* does not always have to lead to traditional *data models*!
5. Know the difference between top-down and bottom-up data modeling approaches and complete a top-down case study.

This book is written for anyone who is working with, or will be working with, MongoDB including business analysts, data modelers, database administrators, developers, project managers, and data scientists. There are three sections:

- In Section I, *Getting Started*, we will reveal the power of data modeling and the important connections to data models that exist when designing any type of database (Chapter 1); compare NoSQL with traditional relational databases and find where MongoDB fits (Chapter 2); explore each MongoDB object and comprehend how each compares to its data modeling and traditional relational database counterparts (Chapter 3); and explain the basics of adding, querying, updating, and deleting data in MongoDB (Chapter 4).
- I n Section II, *Levels of Granularity*, we cover Conceptual Data Modeling (Chapter 5), Logical Data Modeling (Chapter 6), and Physical Data Modeling (Chapter 7). Notice the "ing" at the end of each of these chapters. We focus on the process of building each of these models, which is where we gain essential business knowledge.

- I n Section III, *Case Study*, we will explain both top-down and bottom-up development approaches and complete a top-down case study where we start with conceptual data modeling and end with a MongoDB database. This case study will tie together the conceptual, logical, and physical techniques from Section II.

Key points are included at the end of each chapter as a way to reinforce concepts. In addition, this book is loaded with hands-on exercises along with their answers, which are provided in Appendix A. Appendix B contains all of the book's references, and Appendix C contains a glossary of the terms used throughout the text. There is also a

comprehensive index.

Conventions Used in This Book

We will use the shortcut *RDBMS* for Relational Database Management System. RDBMS represents the traditional relational database invented by E. F. Codd at IBM in 1970 and first commercially available in 1979 (which was Oracle) [Wikipedia]. Popular RDBMS databases today include Oracle, Sybase, Microsoft SQL Server, and Teradata.

We use the Embarcadero ER/Studio® Data Architect tool to build our data models throughout the text. Learn more about this tool at this website: http://www.embarcadero.com/products/er-studio-data-architect.

There is an important distinction between the term *relational database* and the term *relational*. The term *relational database* refers to the technology on how the data is stored, whereas the term *relational* implies the technique of modeling business rules and applying normalization.

The term *object* includes any data model component such as entities, attributes, and relationships. Objects also include any MongoDB component such as fields, documents, and collections.

We make use of the following simple conventions:

Object names	**Customer Last Name** is an attribute of **Customer**.
MongoDB code	db.account.find( )
Object instances	*Bob Smith* is an instance of **Student**.

I am a firm believer in learning through playing.

We might as well have fun learning, so throughout the book, "play" and build your own data models and MongoDB collections. It is very easy to get started with a MongoDB server and client on your computer, as you will see in Chapter 2. I hope you will realize as I do that data modeling and MongoDB go together like peanut butter and jelly!

Section I

Getting Started

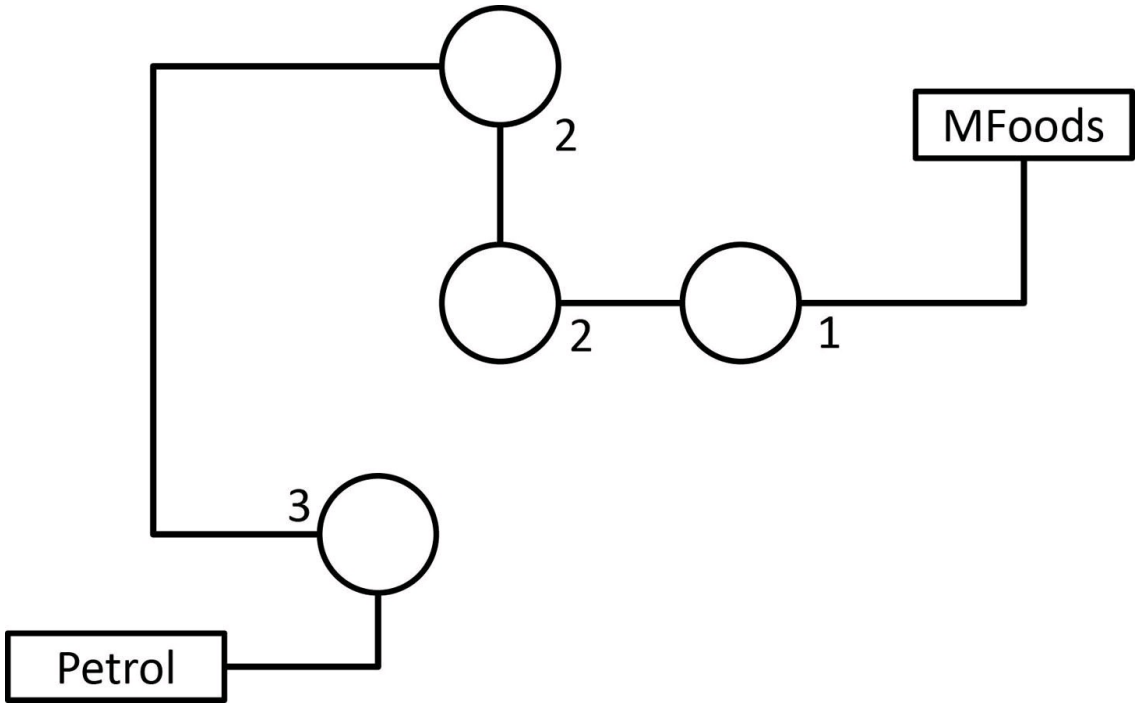
In this section we will reveal the power of data modeling and the important connections to data models that exist when designing any type of database (Chapter 1); compare NoSQL with traditional relational databases and where MongoDB fits (Chapter 2); explore each MongoDB object and comprehend how each compares to its data modeling and traditional relational database counterparts (Chapter 3); and explain the basics of adding, querying, updating, and deleting data in MongoDB (Chapter 4).

By the end of this section, you will know why data modeling is so important to any database, including MongoDB, and be able to explain and put to use the basic set of MongoDB objects and functions. After reading the four chapters in this section, you will be prepared to tackle conceptual, logical, and physical data modeling in Section II.

Chapter 1
The Power of Data Modeling

I gave the steering wheel a heavy tap with my hands as I realized that, once again, I was completely lost. It was about an hour before dawn, I was driving in France, and an important business meeting awaited me. I spotted a gas station up ahead that appeared to be open. I parked, went inside, and showed the attendant the address of my destination.

I don't speak French and the attendant didn't speak English. The attendant did, however, recognize the name of the company I needed to visit. Wanting to help and unable to communicate verbally, the attendant took out a pen and paper. He drew lines for streets, circles for roundabouts along with numbers for exit paths, and rectangles for his gas station and my destination, an organization called "MFoods":



With this custom-made map, which contained only the information that was relevant to me, I arrived at my address without making a single wrong turn. The map was a model of the actual roads I needed to travel.

Now, what does my poor sense of direction and a gas station attendant skilled at drawing maps have to do with MongoDB? A map simplifies a complex *geographic* landscape in the same way that a data model simplifies a complex *information* landscape. In many cases with large data volumes, high velocity in receiving data, and diverse data types, the complexities we encounter in MongoDB applications can make those roundabouts in France look ridiculously simple. We therefore need maps (in the form of data models) to provide clear and precise documentation about an application's information landscape.

It would probably have taken me hours of trial and error to reach my destination in France, whereas that simple map the gas station attendant drew provided me with an almost instantaneous broad understanding of how to reach my destination. A model makes use of standard symbols that allow one to grasp the content quickly. In the map he drew for me, the attendant used lines to symbolize streets and circles to symbolize roundabouts. His skillful use of those symbols helped me visualize the streets and roundabouts.

Data modeling is the process of learning about the data, and the data model is the end result of the data modeling process. A data model is a set of symbols and text that precisely explains a business information landscape. A box with the word "Customer" within it represents the concept of a real **Customer** such as *Bob*, *IBM*, or *Walmart* on a data model. A line represents a relationship between two concepts such as capturing that a **Customer** may own one or many **Accounts**.

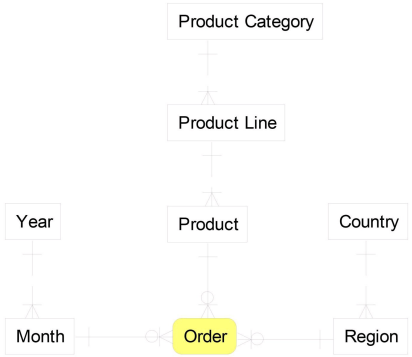
This chapter explains why data modeling is necessary for any database, relational or NoSQL, and also introduces the publishing case study that appears in each of the following chapters.

Many Forms to Represent an Information Landscape

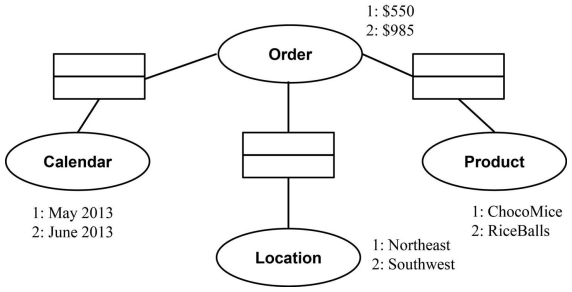
The result of the data modeling process is a data model, yet data models themselves can be represented through many different forms. Data models can look like the box and line drawings that are the subject of this book, or they can take other forms such as Unified Modeling Language (UML) Class Diagrams, spreadsheets, or State Transition Diagrams. They can even take the form of precise business assertions generated from the answers to business questions.

For example, here are four forms of data models:

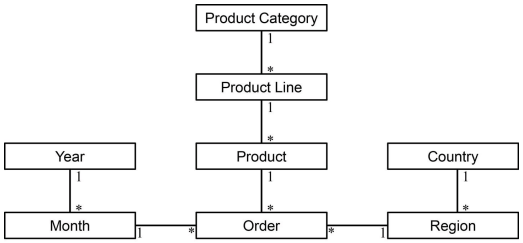
Information Engineering



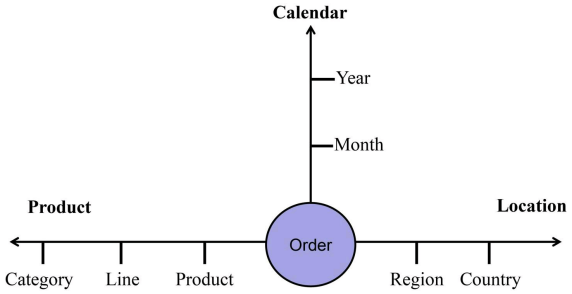
Fully Communication-Oriented Information Modeling



Unified Modeling Language



The Axis Technique

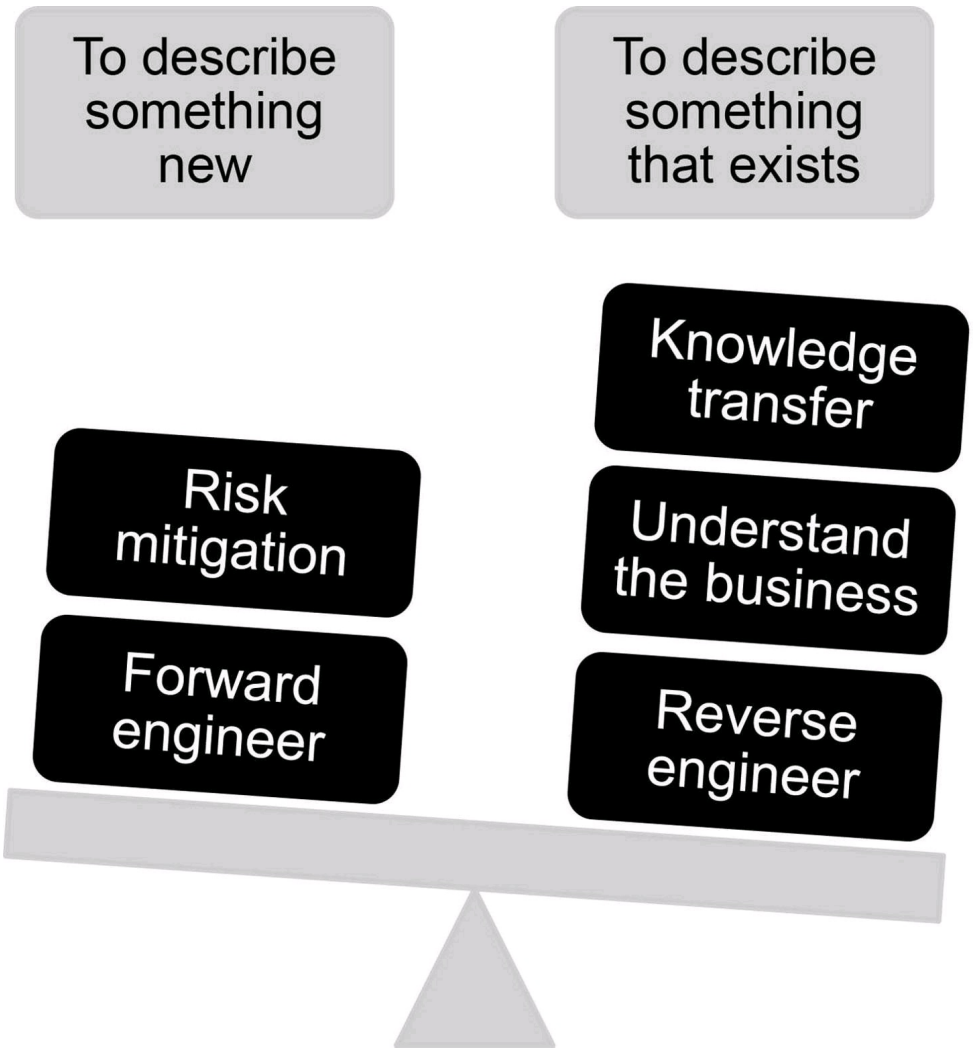


Each depicts the same business area but uses a different set of symbols. Which form works best? It depends on the audience. The data modeler can get very creative with the form used to explain an application's information landscape.

Confirming and Documenting Different Perspectives

The reason we do data modeling is to confirm and document our understanding of different perspectives. A data model is a communication tool. Think of all of the people involved in building even a simple application: business professionals, business analysts, data modelers, data architects, database developers, database administrators, developers, managers, etc. People have different backgrounds and experiences and varying levels of business knowledge and technical expertise. The data model allows us to confirm our knowledge of the area and make sure people see the information landscape similarly or, at a minimum, have an understanding of the differences that exist.

A data model can describe a new information landscape, or it can describe an information landscape that currently exists. This figure contains the new and existing areas where data modeling can be leveraged:



Traditionally, data models have been built during the analysis and design phases of a project to ensure that the requirements for a new application are fully understood and correctly captured before the actual database is created (i.e., forward engineering). There are, however, other uses for modeling than simply building databases. Among these uses are the following:

- **Risk mitigation**. A data model can capture the concepts and interactions that are impacted by a development project or program. What is the impact of adding or modifying structures for an application already in production? One example of impact analysis would be to use data modeling to determine what impact modifying its structures would have on purchased software.
- **Reverse engineer**. We can derive a data model from an existing application by examining the application's database and building a data model of its structures. The technical term for the process of building data models from existing applications is "reverse engineering." The trend in many industries is to purchase

software and to build less internally; therefore, our roles as modelers are expanding and changing. Instead of modeling a new application, the data modeler may capture the information in existing applications.

- **Understand the business**. As a prerequisite to a large development effort, it usually is necessary to understand how the business works before you can understand how the applications that support the business will work. Before building an order entry system, for example, you need to understand the order entry business process. The data and relationships represented in a data model provide a foundation on which to build an understanding of business processes.
- **Knowledge transfer**. When new team members need to come up to speed or developers need to understand requirements, a data model is an effective explanatory medium. Whenever a new person joined our department, I spent some time walking through a series of data models to educate the person on concepts and rules as quickly as possible.

Data Modeling Is Not Optional!

The power of the data model as a tool to confirm and document our understanding of different perspectives has, as the root of its power, one word: *Precision*. Precision, with respect to data modeling, means that there is a clear, unambiguous way of reading every symbol and term on the model. You might argue with others about whether the rule is accurate, but that is a different argument. In other words, it is not possible for you to view a symbol on a model and say, "I see A here" and for someone else to view the same symbol and respond, "I see B here." In a data model, precision is primarily the result of applying a standard set of symbols. The traffic circles the gas station attendant drew for me were standard symbols that we both understood. There are also standard symbols used in data models, as we will discover shortly.

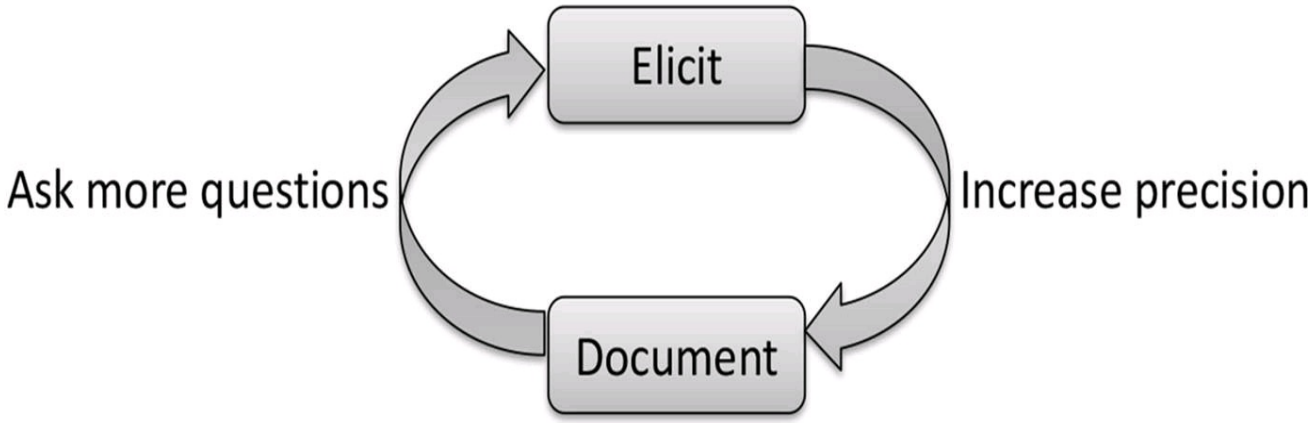
The process of understanding and precisely documenting data is not an optional process. As long as there is at least some data in the application and at least two people involved in building or using the application, there is a need to confirm and document our understanding of their perspectives.

Take **Customer** for example. You can start off with the innocent question, "How do you define **Customer**?" This question is not as easy to answer as it may appear because of the many different perspectives on **Customer**. Once you get your answer, you can move on to other data modeling questions, such as:

- How do you identify a **Customer**?

- How do you describe a **Customer**?
- Can a **Customer** own more than one **Account**?
- Can an **Account** be owned by more than one **Customer**?
- Can a **Customer** exist without owning any **Accounts**?

These are some of the many questions that get asked during the data modeling process. Asking and getting answers to questions like these is called *elicitation*, and data modeling includes eliciting and documenting data requirements. There is an iterative process between eliciting and documenting data requirements:



While data modeling, we ask questions to increase the precision of our data model, and then through the process of documenting our data model, we ask more questions. This loop continues until we are done with our model. A lot of knowledge is gained as the people involved in the modeling process challenge each other about terminology, assumptions, rules, and concepts.

During the process of modeling a recipe management system for a large manufacturing company, I was amazed to witness team members with years of experience debate whether the concept of an **Ingredient** differed from the concept of **Raw Material** after asking the question, "What is the difference between an **Ingredient** and a **Raw Material**?" After a 30-minute discussion on ingredients and raw materials, everyone who participated in this modeling effort benefited from the debate and left the modeling session with a stronger understanding of recipe management. We documented the results on the data model and moved on to asking more questions, continuing this iterative data requirements process.

Embarking on Our Publishing Adventure

I am not only the author of this book, but I also manage the publishing company that published it. So in addition to being a data modeler, I am also the CEO of a large publishing empire called *Technics Publications, LLC*. "Empire" might be a slight exaggeration, as I am also the only employee. Being a one-person company has a very large advantage, however, in that I know the publishing business inside and out. I can describe every step necessary to produce a book. Publishing is a great example to use in this book because I can be your business expert, and together, we will perform the data modeling from idea through MongoDB design.

In fact, let's build our first publishing data model right now! However, instead of the typical boxes and lines we associate with a data model, let's start with a data model in the form of a spreadsheet. Data models come in many forms, and perhaps the most common form of data model we work with on a daily basis is the spreadsheet. A spreadsheet is a representation of a paper worksheet containing a grid defined by rows and columns where each cell in the grid can contain text or numbers. The columns often contain different types of information.

As a publisher, I need to submit information about each new title to many stores and distributors. Businesses such as Amazon.com® and Barnes & Noble® need to know information about an upcoming title so they can sell it on their Websites and in their stores.

Take a moment and look at the front and back covers of this book. What information do you see?

Here is a subset of the information I saw when doing this exercise:

Data Modeling for MongoDB
Steve Hoberman
$39.95

DATABASE / DATA MODELING
9781935504702

We could also list the graphics, barcode, etc., but this is enough to get us started.

We can now take this information and list it in the form of a spreadsheet:

Title Name	Author Name	Title Retail Price	Category Name	ISBN
Data Modeling for MongoDB	Steve Hoberman	$39.95	DATABASE / DATA MODELING	9781935504702

In the first row, we listed the type of information, and in the second row, the values for each type of information. This is a data model because it is representing data using a set of symbols and text – in this case, in the common form of a spreadsheet.

Now there are business questions that need to be answered about the types of information such as:

- Can a **Title** be written by more than one **Author**?
- Can an **Author** write more than one **Title**?
- Can a **Title** exist without an **Author**?
- Can an **Author** exist without a **Title**?

Of course, there is quite a bit more information we could model in this spreadsheet, but in order to keep things readable in the layout on this page, we can stop here for now and confidently conclude that we completed our first data model.

There are three very important similarities between the exercise we just completed on modeling the information on this book and any modeling assignment we do for real:

1. **Process.** The process we went through, where we looked at something

ambiguous (such as a book cover) and brought precision to it, is what the data modeler spends at least half our time working on. Data modeling is the process of organizing things and making them more precise. Data modelers are fantastic organizers. We can take complete chaos and bring clarity by organizing and sorting "things" – in this case, descriptive information about a title becomes neatly organized into a spreadsheet.

2. **Creativity.** There are many ways we can communicate information via a data model through various data modeling notations and formats. You can choose Information Engineering (IE, for short), which is the notation used in this book, Integration Definition for Information Modeling (IDEF1X), Object Role Modeling (ORM), the Unified Modeling Language (UML) Class Diagram, and many others. But the documentation coming out of data modeling does not need to be in the form of a traditional data model – it can also be a list of business assertions, for example. It comes down to knowing which form the audience for the data model would best understand. After all, a data model is a communication tool, and therefore we should choose the visual that is easiest to communicate for a particular audience. We can be very creative on which forms to use in a given situation. In this first publishing example, we have chosen to use the easy-to-understand spreadsheet format.

3. **80/20 rule.** Every modeling assignment I have ever worked on has been constrained by time. I always wish I had more time to complete the data model. "If only I had one more week, I could really make this perfect." In reality, we never reach perfection, and I have learned over the years that in 20% of the time, we can get the data model at least 80% right; getting that last 20% right takes a lot more time to complete and may not be worth the effort anyway. In many assignments, for example, the complete data modeling deliverables include a document listing all of the outstanding issues that came up during the modeling process. This list of issues represents the remaining 20% that, after answering, would get the model closer to perfection. In this book example, you were given just a few moments to think about the information on this cover – that's not a lot of time!

However, there is one very important difference between the model you just completed and the ones we do for real: there is no one around to argue our solution with us! That is, should we call it a "Title" or a "Book"? What's a good definition for **Title** anyway? These types of discussions are often where the data modeler spends quite a bit of time. If the data modeler spends about 50% of our time organizing information, the other 50% can be spent working with different groups to come to consensus on terminology and definitions. I like to call this role of getting people to

see the world the same way, or at least documenting their differences, as the role of *Ambassador*. So half the time we are Organizers and the other half we are Ambassadors – pretty important roles!

EXERCISE 1: LIFE WITHOUT DATA MODELING?

Think about a project you worked on (or inherited from another group to support or enhance) that didn't use data modeling. Was the project successful? Discuss both the short term success factors, such as whether the project was completed on time and within budget, as well as the longer term factors such as whether the project was supportable or was easy to customize or integrate with other applications. Refer to Appendix A for my thoughts.

Key Points

- Data modeling is the process of learning about the data, and the data model is the end result of the data modeling process. A data model is a set of symbols and text to precisely explain a business information landscape.
- The data modeler can get very creative with the form used to explain the information landscape.
- The reason we do data modeling is to confirm and document our understanding of other perspectives. A data model is a communication tool.
- Traditionally, data models have been built during the analysis and design phases of a project to ensure that the requirements for a new application are fully understood and correctly captured before the actual database is created (i.e. forward engineering). There are, however, other uses for modeling than simply building databases including mitigating risk, performing reverse engineering, understanding the business, and transferring knowledge.
- Precision, with respect to data modeling, means that there is a clear, unambiguous way of reading every symbol and term on the model.
- There is an iterative cycle between eliciting and documenting requirements.
- Remember the three important points that came out of our book modeling exercise: 1) it's always the same process of organizing things, 2) be creative on the form used for the data model, and 3) always follow

the 80/20 rule so that you can complete 80% of the data model in 20% of the time.

- Half the time in data modeling we are organizers and the other half we are ambassadors.

Chapter 2
The Power of NoSQL and MongoDB

This chapter explains NoSQL along with a comparison to traditional relational databases. We briefly discuss the four main types of NoSQL databases: document, key-value, column-oriented, and graph. Then we cover what makes MongoDB unique followed by how to install MongoDB on your computer.

NoSQL vs. the Traditional Relational Database

NoSQL is a name for the category of databases built on non-relational technology. NoSQL is not a good name for what it represents as it is less about how to query the database (which is where SQL comes in) and more about how the data is stored (which is where relational structures comes in). Even Carlo Strozzi, who first used the term NoSQL in 1998 to name his lightweight, open-source relational database that did not expose the standard SQL interface, suggests that "NoREL" would have been a better term than NoSQL [Wikipedia]. However, because the term NoSQL is in widespread use today, we will continue to use this term in this chapter.

There are four factors that distinguish traditional relational databases (abbreviated as "RDBMS") from NoSQL databases: variety, structure, scaling, and focus. Here is a table summarizing these differences:

	RDBMS	NoSQL
Variety	One type (relational)	Four main types: document, column-oriented, key-value, and graph

Structure	Predefined	Dynamic
Scaling	Primarily vertical	Primarily horizontal
Focus	Data integrity	Data performance and availability

VARIETY

RDBMS have one type of structure (relational) where individual records are stored as rows in tables, with each column storing a specific piece of data about that record. When data is needed from more than one table, these tables are joined together. For example, offices might be stored in one table and employees in another. When a user wants to find the work address of an employee, the **Employee** and **Office** tables are joined together. NoSQL has four main varieties, and each will be described shortly: document, column-oriented, key-value, and graph. Each variety has its own way of storing data. An RDBMS has traditionally been chosen as the solution for every type of operational or analytical scenario. NoSQL solutions, however, excel in particular scenarios and therefore are used for specific types of situations (e.g., a graph database is often used in scenarios where a document database would be inefficient).

STRUCTURE

The RDBMS database structure is predefined. To store information about a new property, we need to modify the structure before we can add the data. Before we can add all of our employees' birth dates, we first need to create the **Employee Birth Date** field. The NoSQL structures are typically dynamic. New types of information can be added as needed without having to reload data or rebuild database structures.

SCALING

RDBMS are usually just scaled vertically, meaning a single server must be upgraded in order to deal with increased demand. NoSQL databases are usually scaled horizontally, meaning that to add capacity, a database administrator can add more inexpensive servers or cloud instances. The database automatically replicates and/or divides data across servers as necessary. This process of automatically replicating or dividing data is called "sharding" in MongoDB.

FOCUS

With RDBMS, the focus is on data integrity. A handy acronym to remember is ACID (Atomic, Consistent, Isolated, and Durable):

- **Atomic**. Everything within a transaction succeeds or the entire transaction is rolled back. For example, if an **Order** is deleted, all of its **Order Lines** are also deleted. When I transfer $10 from my savings to checking account, $10 is deducted from my savings account and credited to my checking account.
- **Consistent**. Consistent means that the data accurately reflects any changes up to a certain point in time. With RDBMS it is the current state, which is achieved through database constraints. A transaction cannot leave the database in an inconsistent state. If an **Order Line** exists without its **Order**, recreate the **Order** or remove the **Order Line**. If $10 was deducted from my savings account but not credited to my checking account, roll the transaction back so the $10 goes back to my checking account.
- **Isolated**. Transactions cannot interfere with each other. That is, transactions are independent. The $10 transferred from my savings to checking account is a separate transaction from the $20 check I wrote, and these two transactions (fund transfer or fund withdrawal) are separate from each other.
- **Durable**. Completed transactions persist even when servers restart or there are power failures. That is, there is no undo button. Once the **Order** and its **Order Lines** are deleted, they are gone from the system even if electricity is lost in the building in which the server resides.

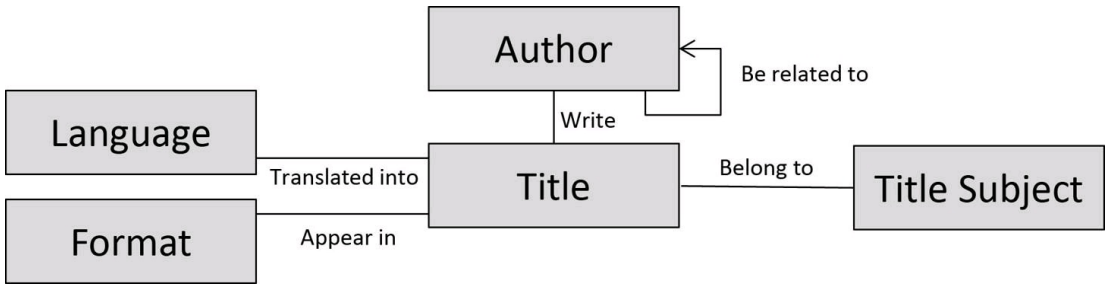
With NoSQL, the focus is less on data integrity and more on data performance and availability. A handy acronym to remember is BASE (Basically Available, Soft-state, Eventual consistency):

- **Basically Available**. There is a response to every query, but it could be a response saying there was a failure in getting the data or the response that comes back may be in an inconsistent or changing state.[1] As an Amazon Seller, for example, when I check my book inventory levels, there is usually a warning in bold red letters saying the data I am viewing is incomplete or more than 24 hours old and therefore may not be accurate.
- **Soft-state**. Soft-state means the NoSQL database plays “catch up,” updating the database continuously even with changes that occurred from earlier in the day. Although no one might have purchased a book from me between 2 and 3 am, for example, there might still be transactions occurring during this time that adjust my inventory levels for purchases from the prior day.

- **Eventual consistency**. Recall that "consistent" means that the data accurately reflects any changes up to a certain point in time[2]. With NoSQL, the system will eventually become consistent once it stops receiving input. It is acceptable for the results not to be 100% current, assuming the accuracy eventually catches up at the end of the week or month. Even though inventory adjustments might occur minutes or hours after book purchases, eventually the inventory levels I am viewing will be accurate.

Four Types of NoSQL Databases

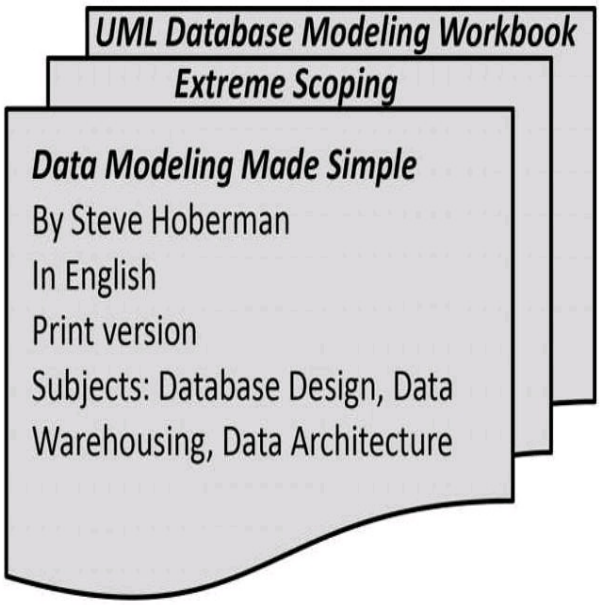
Here is a sketch for the traditional RDBMS structure for storing author-and title-related information:



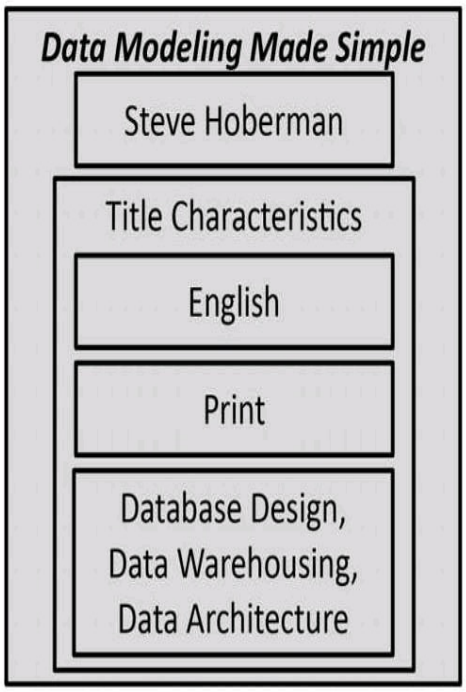
The RDBMS structure stores like information together such as authors in the **Author** table and titles in the **Title** table. The lines connecting these tables represent the rules (also known as "database constraints") that enforce how many values of one table can connect to how many values of the other table. For example, the line between **Author** and **Title** would capture that an author can write many titles and a title can be written by many authors. If we wanted to know who wrote a particular title, we would need to join **Title** with **Author**.

There is also a line from **Author** to **Author**, which captures any relationship between authors such as one author who wrote a book review for another author or two authors who are on the same tennis team, etc. A constraint that starts and ends on the same table, such as this author constraint, is called a recursive relationship. (Recursive relationships will be discussed in the next chapter.)

Let's contrast this traditional way of structuring data with the four varieties of NoSQL databases. Here is how the author and title information can be captured in each of these four types of databases:



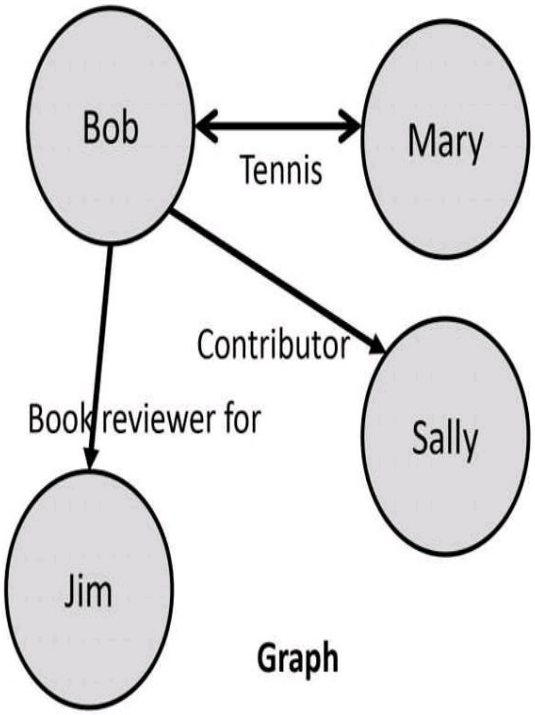
Document-oriented
(Library Card Catalog)



Column-oriented

Key	Value
Title Name	***Data Modeling Made Simple***
Author Name	Steve Hoberman
Language	English
Format	Print version
Subjects	Database Design, Data Warehousing, Data Architecture

Key-value



Graph

- **Document**. Instead of taking a business subject and breaking it up into multiple

relational structures, document databases frequently store the business subject in one structure called a "document." For example, instead of storing title and author information in two distinct relational structures, title, author, and other title-related information can all be stored in a single document called **Title**. This is similar to a search you would do in a library for a particular book, where both title and author information appear together. We have everything to do with the title, including authors and subject, in the same document. It is all in one place, and we do not have to join to separate places to get everything we need. Document-oriented is much more application focused as opposed to table oriented, which is more data focused. MongoDB is a document-based database.

- **Key-value**. Key-value databases allow the application to store its data in only two columns ("key" and "value"), with more complex information sometimes stored within the "value" columns such as **Subjects** in our example above. So instead of doing the work ahead of time to determine whether we have authors or titles, we can add each type of information (**Title Name**, **Author Name**, etc.) as a key and then add the value assigned to it. Key-value databases include Dynamo, Cache, and Project Voldemort.

- **Column-oriented**. Out of the four types of NoSQL databases, column-oriented is closest to the RDBMS. Both have a similar way of looking at data as rows and values. The difference, though, is that RDBMSs work with a predefined structure and simple data types, such as amounts and dates, whereas column-oriented databases, such as Cassandra, can work with more complex data types including unformatted text and imagery. This data can also be defined on the fly. So in our author/title example, we can have a title and the title has an author along with a complex data type called **Title Characteristics**, which contains the language, format, and subjects, where subjects is another complex data type (an array).

- **Graph**. This kind of database is designed for data whose relations are well represented as a set of nodes with an undetermined number of connections between these nodes. Examples where a graph database can work best are social relations (where nodes are people), public transport links (where nodes could be bus or train stations), or road maps (where nodes could be street intersections or highway exits). Often requirements lead to traversing the graph to find the shortest routes, nearest neighbors, etc., all of which can be complex and time consuming to navigate with a traditional RDMBS (usually requiring recursion, which is not for the faint of heart and will be discussed in the next chapter). In our author example above, there are an indeterminate number of ways authors can

be related with each other such as contributors, co-authors, book reviewers, etc. Graph databases include Neo4J, Allegro, and Virtuoso.

MongoDB Is a Document-Oriented NoSQL Database

Since you're reading this book, you've probably already chosen to implement your database in MongoDB. MongoDB is known for high performance, high availability, and low cost because of these four properties:

1. **Document-oriented**. Instead of taking a business subject and breaking it up into multiple relational structures, MongoDB can store the business subject in the minimal number of doucments. For example, instead of storing title and author information in two distinct relational structures, title, author, and other title-related information can all be stored in a single document called **Book**, which is much more intuitive and usually easier to work with.
2. **Extremely extensible**. The document-oriented approach makes it possible to represent complex hierarchical relationships in one place. You do not need to define schemas ahead of time; instead, you can define new types of data (called "fields") as you add the actual data. This is very different from building traditional relational databases where you need to define the structure before it is populated with data. In MongoDB, you can populate and define the structure at the same time. This makes development take less time and allows the project team to easily experiment with different solutions and then choose the best one. The figure below illustrates that with an RDBMS, you need to predefine the structure. In music, you need to write down the musical notes before you can play them. With MongoDB, however, you can define the structure of data and populate the data at the same time, much like composing music and playing it at the same time.

RDBMS

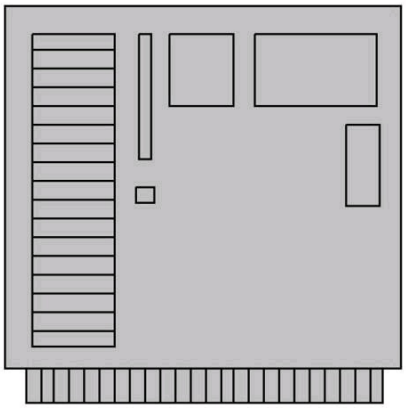
Plan and then play

MongoDB

Play and plan at the same time

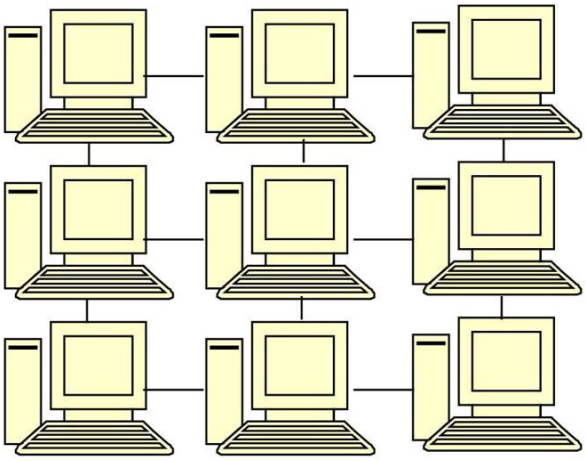
1. **Horizontally scalable**. MongoDB was designed to scale out. Documents are relatively easy to partition across multiple servers. MongoDB can automatically balance data across servers and redistribute documents, automatically routing user requests to the correct machines. When more capacity is needed, new machines can be added, and MongoDB will figure out how the existing data should be allocated to them. This figure illustrates the systems architecture difference:

RDBMS



Scale by upgrading box

MongoDB



Scale by adding inexpensive servers

1. **Lots of drivers!** A driver is a translator between a program and a platform (the platform in this case being MongoDB). MongoDB has an official set of drivers

whose core functions, such as find( ) and update( ), are described in Chapter 4. In addition, the development community has many other drivers for various languages support such as for Ruby, Python, and C++. The full list of drivers appears at http://docs.mongodb.org/ecosystem/drivers/. This allows developers to use their language of choice and leverage their existing skills to build MongoDB applications even faster and more efficiently.

Installing MongoDB

The MongoDB installation guide, http://docs.mongodb.org/manual/installation/, has well-written instructions on how to install MongoDB on the various operating systems. I followed these instructions and downloaded the Windows executable from http://www.mongodb.org/downloads. After the file finished downloading, I double-clicked on the executable and followed the steps for installation.

After installation was complete, I set up a default directory where all of the data can be stored. To do this, I right-clicked on the command prompt icon and chose "Run as Administrator," and then typed this statement to set up the default directory:

```
mkdir -p \data\db\
```

Once you set up your default directory, you can start the MongoDB server. If you still have the command prompt window open after creating the default directory, type in this statement to start the MongoDB server:

```
C:\mongodb\bin\mongod.exe
```

If you need to open the command prompt window, always right-click on the command prompt icon and choose "Run as Administrator." I ran the above statement with no parameters, and therefore the MongoDB server will use the default data directory, \data\db\.

You can run both the client and server on the same machine. To run a client, just open another command prompt window (also with "Run as Administrator") and initiate the client by typing:

```
C:\mongodb\bin\mongo.exe
```

You now have both your MongoDB server and client running on your machine! This whole process probably took ten minutes. Amazingly easy.

A great resource for learning MongoDB is https://university.mongodb.com.

Note that the MongoDB shell is a full-featured JavaScript interpreter. You can therefore run any JavaScript commands such as:

```
> y = 100
100
> y / 20
5
```

You can also use all of the standard JavaScript libraries and functions.

When your MongoDB statement spans more than one line, press <Enter> to go to the next line. The MongoDB client knows whether the statement is complete or not, and if the statement is not complete, the client will allow you to continue writing it on the next line. Pressing <Enter> three times will cancel the statement and get you back to the > prompt.

To stop the MongoDB server, press Ctrl-C in the window that is running the server. To stop the MongoDB client, press Ctrl-C in the window that is running the client.

Key Points

- NoSQL is a name for the category of databases built on non-relational technology.
- There are four main differences between traditional relational databases and NoSQL databases: variety, structure, scaling, and focus.
- With RDBMS, the focus is on data integrity. With NoSQL, the focus is on data performance and availability.
- Document databases frequently store the business subject in one structure called a “document.”
- Key–value databases allow the application to store its data in only two columns (“key” and “value”), with more complex information sometimes stored within the “value” columns.
- Column-oriented databases work with more complex data types such as unformatted text and imagery, and this data can also be defined on the fly.
- A graph database is designed for data whose relations are well represented as a set of nodes with an undetermined number of connections between those nodes.
- MongoDB is known for high performance, high availability, and low cost, because of these four properties: document-oriented, extremely extensible, horizontally scalable, and the availability of many drivers.

- It is easy to install MongoDB; both the client and server can be installed on the same machine.

- A great site for learning MongoDB is https://university.mongodb.com.

Chapter 3
MongoDB Objects

This chapter will cover the core set of data modeling concepts along with their MongoDB counterparts. We will start off with a comparison of data model, relational database, and MongoDB terminology. We will then dive into the core set of MongoDB objects, each explained within its data modeling context. Objects include documents, collections, fields, datatypes, and indexes. We will also show how to handle relationships, hierarchies, and recursion in MongoDB. This table compares terms used for each object across the data model, RDBMS, and MongoDB. We will discuss each of these terms within this chapter.

Data Model	RDBMS	MongoDB
Entity instance	Record or Row	Document
Entity	Table	Collection
Attribute or Data element	Data element, Field or Column	Field
Attribute value or Data element value	Field	Field value

Data element value	value, or Column value	
Format domain	Datatype	Datatype
Relationship	Constraint	Captured but not enforced either through a reference, which is similar to a foreign key, or through embedded documents.
Candidate key (Primary or Alternate key)	Unique index	Unique index
Surrogate key	Globally Unique Id (GUID)	ObjectId
Foreign key	Foreign key	Reference
Secondary key, Inversion entry	Secondary key	Non-unique index
Subtype	Rolldown, Rollup, Identity	Rolldown, Rollup, Identity

Entities

An entity represents a collection of information about something that the business deems important and worthy of capture. A noun or noun phrase identifies a specific entity. An entity fits into one of six categories: who, what, when, where, why, or how. Here is a definition of each of these entity categories along with examples.

Category	Definition	Examples

Who	Person or organization of interest to the enterprise. That is, "*Who* is important to the business?" Often a *Who* is associated with a role such as Customer or Vendor.	Employee, Patient, Player, Suspect, Customer, Vendor, Student, Passenger, Competitor, Author
What	Product or service of interest to the enterprise. It often refers to what the organization makes that keeps it in business. That is, "*What* is important to the business?"	Product, Service, Raw Material, Finished Good, Course, Song, Photograph, Title
When	Calendar or time interval of interest to the enterprise. That is, "*When* is the business in operation?"	Time, Date, Month, Quarter, Year, Semester, Fiscal Period, Minute
Where	Location of interest to the enterprise. Location can refer to actual places as well as electronic places. That is, "*Where* is business conducted?"	Mailing Address, Distribution Point, Website URL, IP Address
Why	Event or transaction of interest to the enterprise. These events keep the business afloat. That is, "*Why* is the business in business?"	Order, Return, Complaint, Withdrawal, Deposit, Compliment, Inquiry, Trade, Claim
How	Documentation of the event of interest to the enterprise. Documents record the events such as a Purchase Order recording an Order event. That is, "*How* does the business keep track of events?"	Invoice, Contract, Agreement, Purchase Order, Speeding Ticket, Packing Slip, Trade Confirmation

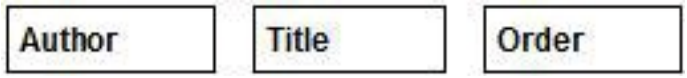
Entity instances are the occurrences or values of a particular entity. Think of a spreadsheet as being an entity where the column headings represent the pieces of information about the entity. Each spreadsheet row containing the actual values represents an entity instance. The entity **Customer** may have multiple customer instances with the names *Bob*, *Joe*, *Jane*, and so forth. The entity **Account** can have instances of *Bob's checking account*, *Bob's savings account*, *Joe's brokerage account*, and so on.

Entities can exist at conceptual, logical, and physical levels of detail. We will go into detail into conceptual, logical, and physical modeling in Section II of this book. A short definition, however, of each level will be needed for you to benefit from the

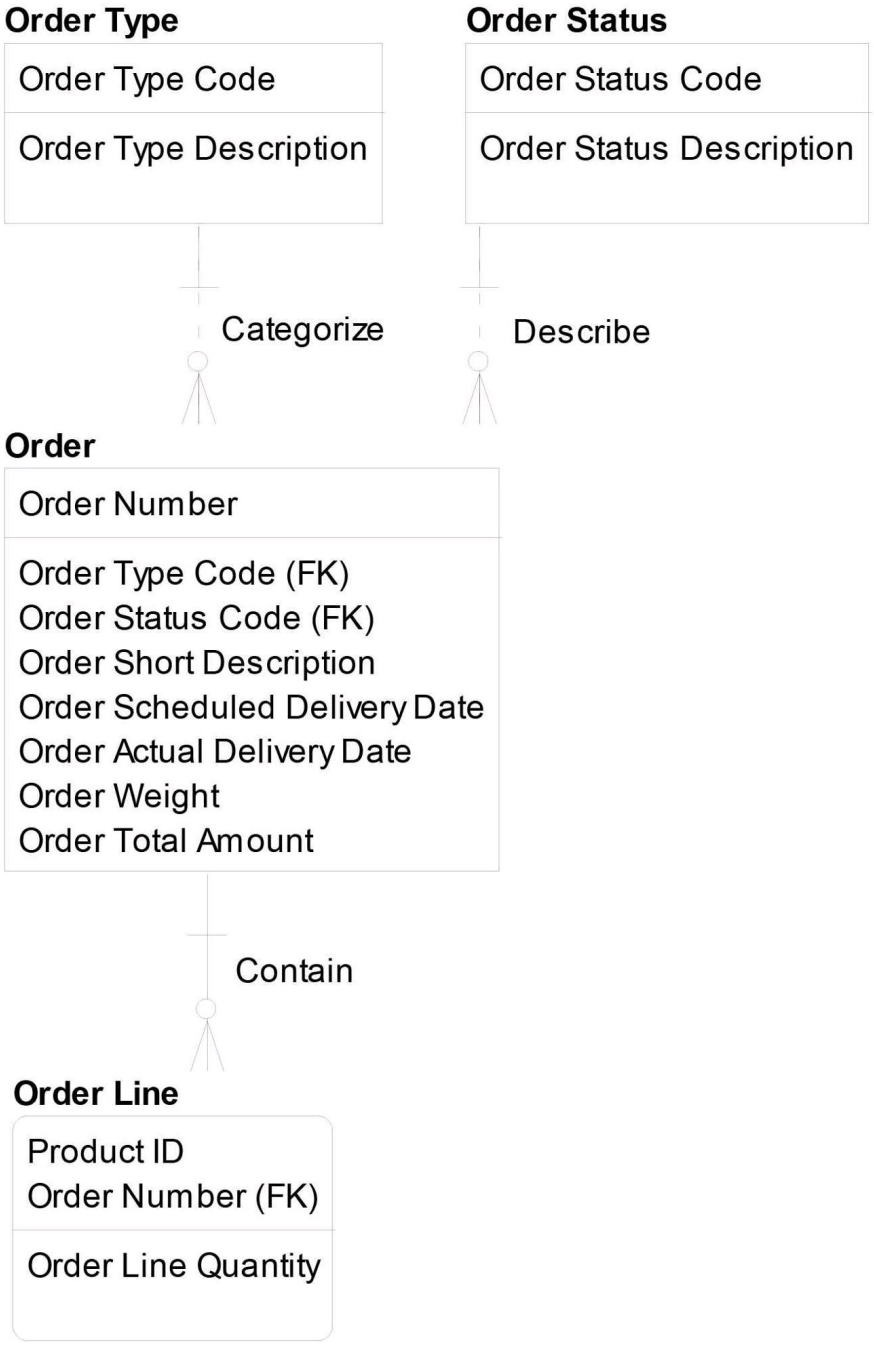
following discussion. The conceptual means the high level business solution to a problem frequently defining scope and important terminology, the logical means the detailed business solution to a problem, and the physical means the detailed technical solution to a problem. “Problem” usually refers to an application development effort.

For an entity to exist at a conceptual level, it must be both basic and critical to the business. What is basic and critical depends very much on the scope of the effort we are modeling. At a universal level, there are certain concepts common to all companies such as **Customer**, **Product**, and **Employee**.

Making the scope slightly narrower, a given industry may have certain unique concepts. **Phone Number**, for example, will be a valid concept for a telecommunications company but perhaps not for other industries such as manufacturing. In the publishing world, **Author**, **Title**, and **Order** are conceptual entities, shown as names within rectangles:



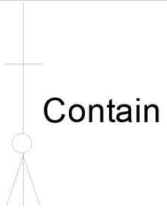
Entities at the logical level represent the business in more detail than at the conceptual level. Frequently, a conceptual entity may represent many logical entities. Logical entities contain properties, also called “data elements” or “attributes,” which we will discuss shortly. Entities are connected through relationships, which we will discuss shortly as well. The figure on the next page shows the **Order** logical entities (along with their properties and relationships) based upon the **Order** conceptual entity from above.



At a physical level, the entities correspond to technology-specific objects, often database tables in a relational database or collections in MongoDB, which we will discuss shortly. The physical level is the same as the logical level, compromised for a specific technology. This model shows the two corresponding physical entities representing relational database tables, based upon the prior logical order entities:

Order

orderNumber (PK)	CHAR(10)	NOT NULL
orderShortDescription	CHAR(10)	NULL
orderScheduledDeliveryDate	DATE	NOT NULL
orderActualDeliveryDate	DATE	NULL
orderWeight	NUMBER(10,2)	NULL
orderTotalAmount	NUMBER(10,2)	NOT NULL
orderTypeCode	CHAR(2)	NOT NULL
orderTypeDescription	CHAR(50)	NULL
orderStatusCode	CHAR(1)	NOT NULL
orderStatusDescription	CHAR(50)	NULL



Order_Line

productID (PK)	CHAR(6)	NOT NULL
orderNumber (PK)(FK)	CHAR(10)	NOT NULL
orderLineQuantity	NUMBER(8,0)	NULL

The physical entities also contain database-specific information, such as the format and length of an attribute (**orderStatusDescription** is 50 characters), and whether the data element is required to have a value (**orderNumber** is not null and is therefore required to have a value, but **orderActualDeliveryDate** is null and therefore not required to have a value).

An entity instance is an example of an entity. So *Order 4839-02* is an instance of **Order**, for example. An entity instance at the RDBMS level is called a record.

MONGODB DOCUMENT = RDBMS RECORD (ENTITY INSTANCE AT PHYSICAL LEVEL)

A RDBMS record is comparable to a MongoDB document. A MongoDB document is a perfect name for what it is – a document. Think of all of the documents you come into contact with during a typical day: invoices, packing slips, menus, blog postings, receipts, etc. All of these are documents in the same sense as a MongoDB document. They are all a set of somewhat related data often viewed together. Documents are composed of fields, which will be discussed shortly. For example, here is a MongoDB document based upon our **Order** example (Order is one document containing three lines):

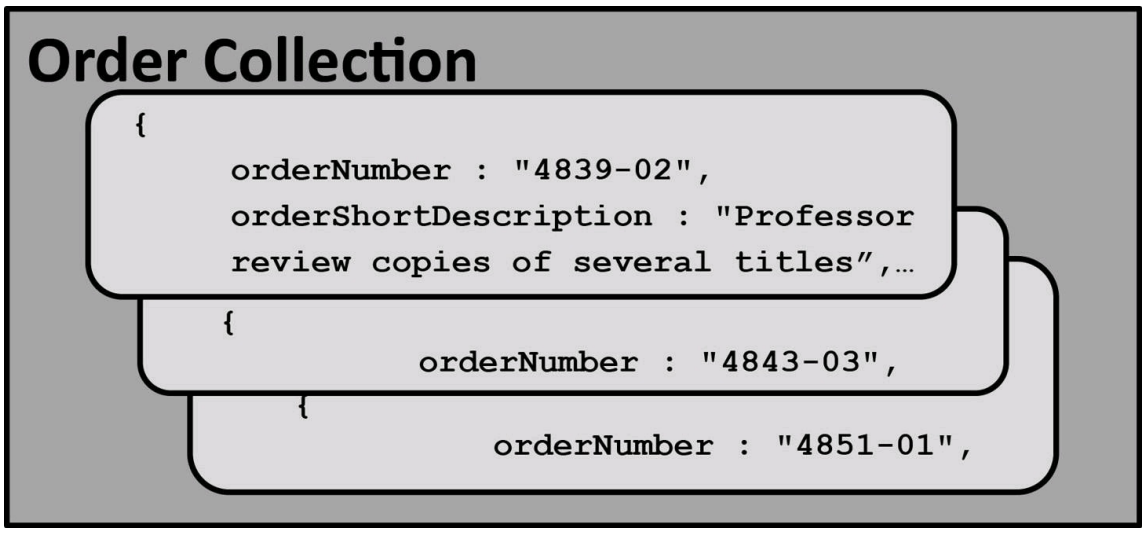
Order:

```
{  orderNumber : “4839-02”,
                    orderShortDescription : “Professor review copies of several titles”,
  orderScheduledDeliveryDate : ISODate(“2014-05-15”),
  orderActualDeliveryDate : ISODate(“2014-05-17”),
  orderWeight : 8.5,
  orderTotalAmount : 19.85,
  orderTypeCode : “02”,
  orderTypeDescription : “Universities Sales”,
  orderStatusCode : “D”,
  orderStatusDescription : “Delivered”,
  orderLine :
   [ {  productID : “9781935504375”,
     orderLineQuantity : 1
   },
                                    {  productID : “9781935504511”,
     orderLineQuantity : 3
   },
   {  productID : “9781935504535”,
     orderLineQuantity : 2
   } ] }
```

In this example the **Order** document contains one order with three order lines. A document begins and ends with the squiggly braces { }, and the fields within the document are separated by commas. The square brackets [ ] contain arrays. Arrays can contain individual values or sub-documents, themselves surrounded by squiggly braces. Storing dates such as **orderScheduledDeliveryDate** in ISO format allows you to use standard date functionality such as performing range queries, sorting dates, and even determining the day of the week.[3] Notice that in **OrderLine**, there is a reference back to the three products. More on references shortly!

MONGODB COLLECTION = RDBMS TABLE (ENTITY AT PHYSICAL LEVEL)

An entity at the physical level is a table in an RDBMS or a collection in MongoDB. A collection is a set of one or more documents. So imagine if we had a million orders, with an example being the one from the prior section. We can store all of these orders in one **Order** collection:



From my relational experience, I am used to defining what the structure would look like first before populating it. For example, I would define the order attributes like **orderNumber** and **orderShortDescription** before loading any orders. In MongoDB however, you can define the structure and data at the same time. Having a "flexible schema" (also known as a "dynamic schema") means incremental changes can be made to the database structure as easily as adding new data. Having such a flexible schema is a big advantage because it lets us add things we might have forgotten or not known about earlier and also because we can test different structures very easily and pick the best one.

Attributes

An attribute is an elementary piece of information of importance to the business that identifies, describes, or measures instances of an entity. The attribute **Claim Number** identifies each claim. The attribute **Student Last Name** describes the last name of each student. The attribute **Gross Sales Amount** measures the monetary value of a transaction.

As with entities, attributes can exist at conceptual, logical, and physical levels. An attribute at the conceptual level must be a concept both basic and critical to the business. We do not usually think of attributes as concepts, but depending on the business need, they can be. When I worked for a telecommunications company, **Telephone Number** was an attribute that was so important to the business that it was represented on a number of conceptual data models.

An attribute on a logical data model represents a business property. Each attribute shown contributes to the business solution and is independent of any technology including software and hardware. For example, **Author Last Name** is an attribute because it has business significance regardless of whether records are kept in a paper file, within Oracle, or within MongoDB.

An attribute on a physical data model represents a database column or MongoDB field. The attribute **Author Last Name** might be represented as the column **AUTH_LAST_NM** within the relational table **AUTH** or the MongoDB field **authorLastName** within the **Author** collection.

MONGODB FIELD = RDBMS FIELD (ATTRIBUTE AT PHYSICAL LEVEL)

The concept of a physical attribute (also called a column or field) in relational databases is equivalent to the concept of a field in MongoDB. MongoDB fields contain two parts, a field name and a field value. This is the **order** collection we discussed earlier, with the field names shown in bold and the field values shown in italics:

```
Order:
 { orderNumber : "4839-02",
        orderShortDescription : "Professor review copies of several titles",
  orderScheduledDeliveryDate : ISODate("2014-05-15"),
  orderActualDeliveryDate : ISODate("2014-05-17"),
  orderWeight : 8.5,
  orderTotalAmount : 19.85,
  orderTypeCode : "02",
  orderTypeDescription : "Universities Sales",
  orderStatusCode : "D",
  orderStatusDescription : "Delivered",
  orderLine :
   [ { productID : "9781935504375",
    orderLineQuantity : 1
   },
                          { productID : "9781935504511",
    orderLineQuantity : 3
   },
   { productID : "9781935504535",
    orderLineQuantity : 2
   } ] }
```

Notice that a field value is not limited to just a simple value such as *4839-02,* but can also include arrays that can contain many other fields and even documents, as we see in this example with **orderLine**. MongoDB provides robust capability for querying within arrays.

Every database has certain restrictions on naming fields, and MongoDB is no

exception. Here are several tips in naming MongoDB fields:

- Avoid special characters. Characters such as a period, dollar sign, or null (\0) should be avoided in field names.
- MongoDB is case sensitive. **CustomerLastName** and **customerLastName** are distinct fields. Make sure your organization has a naming standard so you use case consistently.
- Duplicate names are not allowed. The same document cannot contain two or more of the same field names at the same level of depth. Just like a relational database table cannot contain the element **customerLastName** twice, a MongoDB document cannot contain the same field twice at the same level of depth. You can, however, have the same field name at a different level of depth. For example, although { customerLastName : "Smith", customerLastName : "Jones" } is not allowed, { customerLastName : { customerLastName : "Jones" } } is allowed.

Domains

The complete set of all possible values that an attribute contains is called a domain. An attribute can never contain values outside of its assigned domain, which is defined by specifying the actual list of values or a set of rules. **Employee Gender Code**, for example, can be limited to the domain of (*female*, *male*).

In relational databases, there are three main types of domains:

- **Format**. A format domain restricts the length and type of the data element such as a Character(15) format domain limiting the possible values of a data element to at most 15 characters. A date format limits the values of a data element to valid dates, and an integer format limits the values to any possible integer.
- **List**. A list domain is more restrictive than a format domain and limits the values to a specified defined set such as *male* or *female* or *(A,B,C,D)*.
- **Range**. A range domain restricts the data element values to any value between two other values such as a start and end dates.

MONGODB DATATYPE = RDBMS DATATYPE (FORMAT DOMAIN AT PHYSICAL LEVEL)

A format domain is called a datatype in an RDBMS and also in MongoDB. In MongoDB, there are only datatype domains (no list or range domains). Here are the

main datatype domains:

Format	Description	Example
Boolean	Only the values *1* (also known as *True*) or *0* (also known as *False*)	{ studentAlumniIndicator : Boolean }
Number	MongoDB does not distinguish all of the different number varieties present in a RDBMS. The MongoDB shell defaults to 64-bit floating point numbers.	{ meaningOfLife : 42.153 } or { meaningOfLife : 42 }
String	Any string of UTF-8 characters.	{ meaningOfLife : "forty two" }
Date	Dates are captured as milliseconds since the epoch.	{ orderEntryDate : new Date( ) }
Array	Arrays can be ordered (such as for queues or lists) or unordered (such as for sets). Also, arrays can contain different data types as values and even contain other arrays.	{ author : [“Bill Jones”, “Tom Hanks”, “Edward Scissorhands[1]”] }
Globally Unique Id	Can be used as the primary key value for records and documents. An object id is a 12-byte surrogate key for documents.	{ orderId : ObjectId( ) }

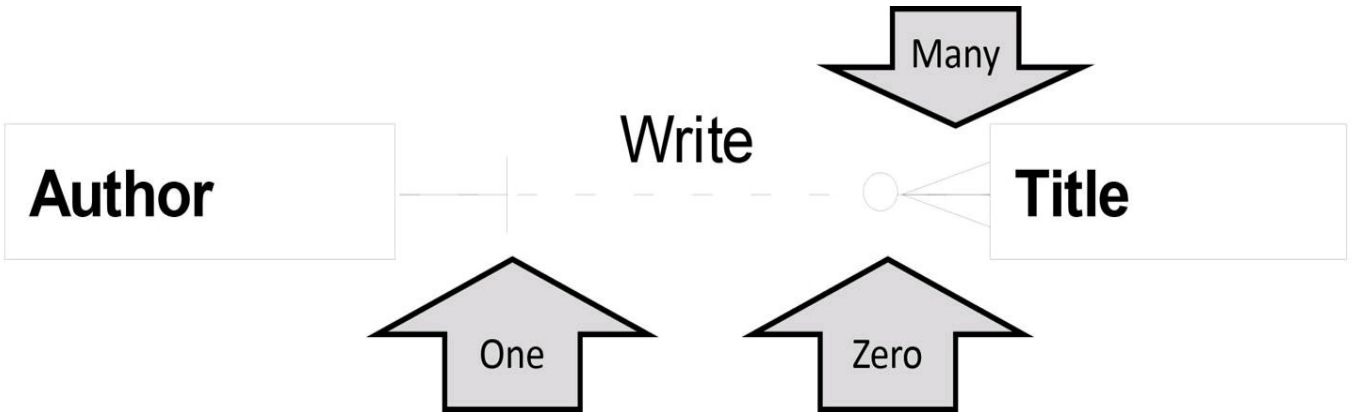
Relationships

Many such rules can be visually captured on our data model through relationships. A relationship is displayed as a line connecting two entities that captures the rule or navigation path between them. If the two entities are **Employee** and **Department**, the relationship can capture the rules "Each **Employee** must work for one **Department**" and "Each **Department** may contain one or many **Employees**."

In a relationship between two entities, cardinality captures how many instances from one entity participate in the relationship with instances of the other entity. It is represented by the symbols that appear on both ends of a relationship line. It is through cardinality that the data rules are specified and enforced. Without cardinality, the most we can say about a relationship is that two entities are connected in some way through a rule. For example, **Employee** and **Department** have some kind of relationship, but we don't know more than this.

For cardinality, we can choose any combination of zero, one, or many. *Many* (some people read it as *more*) means any number greater than zero. Specifying zero or one allows us to capture whether or not an entity instance is *required* in a relationship. Specifying one or many allows us to capture *how many* of a particular instance participates in a given relationship.

Because we have only three cardinality symbols, we can't specify an exact number[4] (other than through documentation), as in "A **Car** contains four **Tires**." We can only say, "A **Car** contains many **Tires**." Each of the cardinality symbols is illustrated here with **Author** and **Title**:



The business rules in this example are:

- Each **Author** may write one or many **Titles**.
- Each **Title** must be written by one **Author**.

The small vertical line means *one*. (Looks like a 1, doesn't it?) The circle means *zero*.

(Looks like a zero too!) The zero implies optionality and does not exclude the value *one*, so in the above example an author can write just one title, too.

The triangle with a line through the middle means *many*. Some people call the *many* symbol a *crow's foot*. Relationship lines are frequently labeled to clarify the relationship and express the rule that the relationship represents. A data model is a communication tool, and if you think of the entities as nouns, the relationship label is a present tense verb, so we are just reading a sentence:

- Each **Author** may write one or many **Titles**.

Having a zero in the cardinality makes us use optional-sounding words such as *may* or *can* when reading the relationship. Without the zero, we use mandatory-sounding terms such as *must* or *have to*. So instead of being redundant and saying:

- Each **Author** may write *zero*, one or many **Titles**.

We take out the word *zero* because it is expressed using the word *may*, which implies the zero:

- Each **Author** may write one or many **Titles**.

Every relationship has a parent and a child. The parent entity appears on the *one* side of the relationship, and the child appears on the *many* side of the relationship. In this example, the parent entity is **Author** and the child entity is **Title**. When I read a relationship, I start with the entity on the *one* side of the relationship (the parent entity) first. "Each **Author** may write one or many **Titles**." It's then followed by reading the relationship from the many side: "Each **Title** must be written by one **Author**."

I also always use the word *each* in reading a relationship, starting with the parent side. The reason for the word *each* is that you want to specify, on average, how many instances of one entity relate to an entity instance from the other entity.

Let's change the cardinality and now allow a **Title** to be written by more than one **Author**:

This is an example of a many-to-many relationship, in contrast to the previous

example, which was a one-to-many relationship. The business rules here read as follows:

- Each **Author** may write one or many **Titles**.
- Each **Title** must be written by one or many **Authors**.

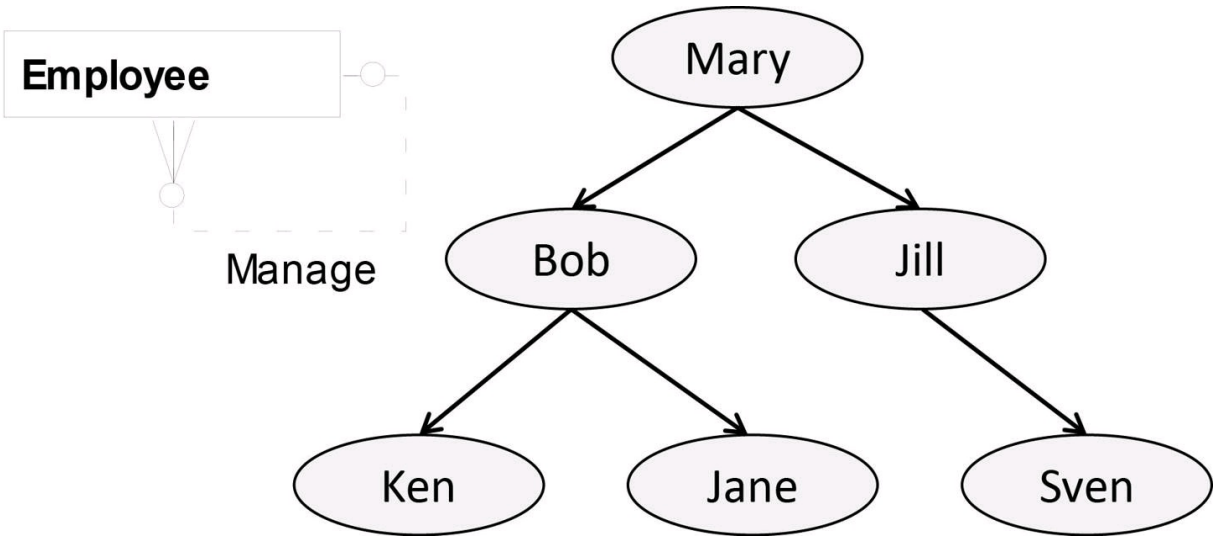
Write, in both of our examples, is an example of a relationship label.

The three levels of granularity (conceptual, logical, and physical) that apply to entities and attributes also apply to the relationships that connect entities. Conceptual relationships are high level rules that connect key concepts. Logical relationships are detailed business rules that enforce the rules between the logical entities. Physical relationships are detailed technology-dependent rules between the physical structures that the relationship connects. These physical relationships eventually become database constraints, which ensure that data adheres to the rules. A "constraint" is a physical term for a relationship in an RDBMS, similar to an entity becoming a table and an attribute becoming a column. MongoDB can capture but not enforce relationships through referencing, or can combine entities together (similar to denormalizing) through embedding. Referencing and embedding will be discussed shortly.

RECURSION

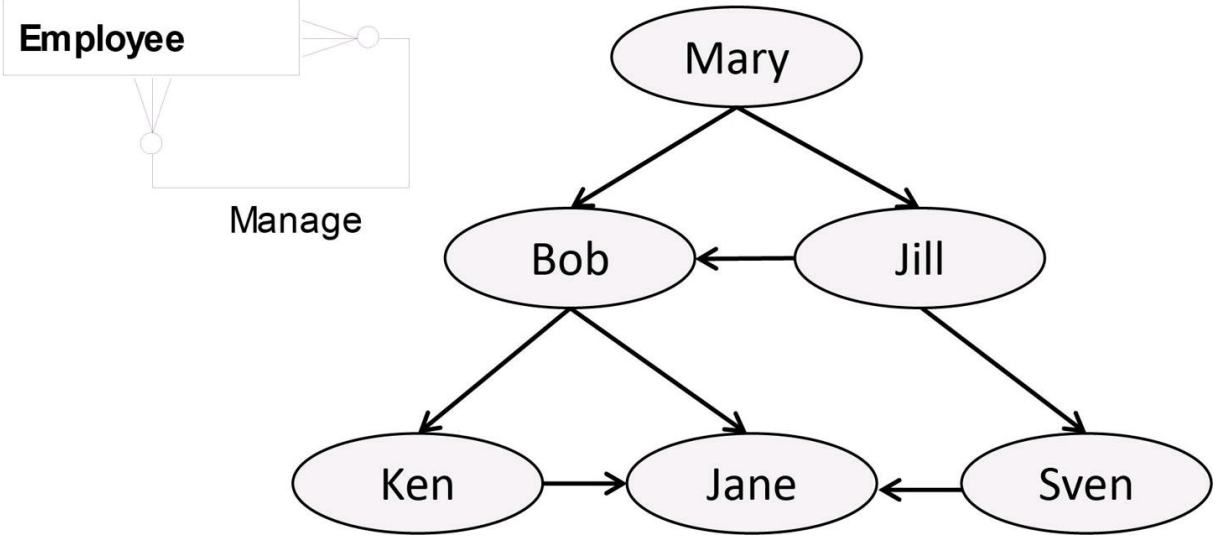
A recursive relationship is a relationship that starts and ends on the same entity. A one-to-many recursive relationship describes a hierarchy, whereas a many-to-many relationship describes a network. In a hierarchy, an entity instance has, at most, one parent. In a network, an entity instance can have more than one parent.

Let's illustrate both types of recursive relationships using **Employee**. Here is a one-to-many recursive example:



- Each Employee may manage one or many Employees.
- Each Employee may be managed by one Employee.

And here is a many-to-many recursive example:



- Each Employee may manage one or many Employees.
- Each Employee may be managed by one or many Employees.

Using sample values such as *Bob* and *Jill* and sketching a hierarchy or network can really help you to understand, and therefore validate, cardinality. The one-to-many captures a hierarchy where each employee has at most one manager. The many-to-many captures a network in which each employee may have one or many managers such as *Jane* working for *Bob*, *Ken*, and *Sven*. (I would definitely update my resume if

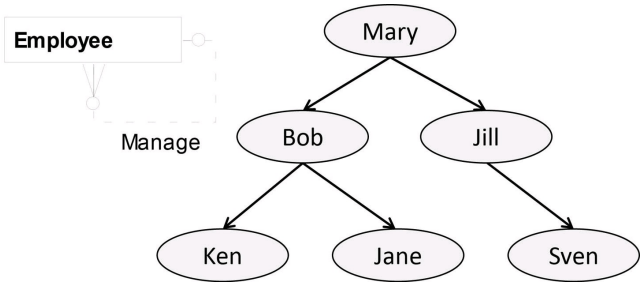
I were *Jane*.)

It is interesting to note that in both figures, there is optionality on both sides of the relationship. In these examples, it implies we can have an **Employee** who has no boss (such as *Mary*) and an **Employee** who is not a manager (such as *Jane*).

Data modelers have a love-hate relationship with recursion. On the one hand, recursion makes modeling a complex business idea very easy and leads to a very flexible modeling structure. We can have any number of levels in an organization hierarchy on both of these models, for example. On the other hand, some consider using recursion to be taking the easy way out of a difficult modeling situation. There are many rules that can be obscured by recursion. For example, where is the *Regional Management Level* on these models? It is hidden somewhere in the recursive relationship. Those in favor of recursion argue that you may not be aware of all the rules and that recursion protects you from having an incomplete model. The recursion adds a level of flexibility that ensures that any rules not previously considered are also handled by the model. It is therefore wise to consider recursion on a case-by-case basis, weighing obscurity against flexibility.

MongoDB Parent Reference = RDBMS Self Referencing (Recursion at Physical Level)

Using the same sample employee data as above, let's look at MongoDB collections for each of these hierarchies and networks. Note that we used the **employeeFirstName** as the unique index and we are only storing the employee's first name (and sometimes manager references), but it will illustrate how MongoDB handles recursion. Here is a collection showing the hierarchy example using parent references:



Employee :

```
{ _id : "Mary", employeeFirstName : "Mary" },
{ _id : "Bob", employeeFirstName : "Bob", managerID : "Mary" },
{ _id : "Jill", employeeFirstName : "Jill", managerID : "Mary" },
{ _id : "Ken", employeeFirstName : "Ken", managerID : "Bob" },
{ _id : "Jane", employeeFirstName : "Jane", managerID : "Bob" },
{ _id : "Sven", employeeFirstName : "Sven", managerID : "Jill" }
```

Depending on the business requirements, we may decide to store the full management

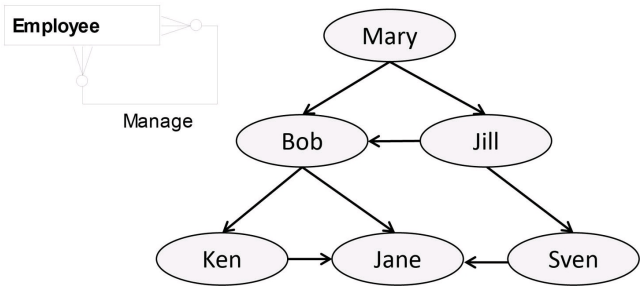
chain with each employee as an array:

Employee :

```
{ _id : “Mary”, employeeFirstName : “Mary” },
{ _id : “Bob”, employeeFirstName : “Bob”, managerID : “Mary”, managerPath : [ “Mary” ] },
{ _id : “Jill”, employeeFirstName : “Jill”, managerID : “Mary”, managerPath : [ “Mary” ] },
{ _id : “Ken”, employeeFirstName : “Ken”, managerID : “Bob”, managerPath : [ “Mary”, “Bob” ] },
{ _id : “Jane”, employeeFirstName : “Jane”, managerID : “Bob”, managerPath : [ “Mary”, “Bob” ] },
{ _id : “Sven”, employeeFirstName : “Sven”, managerID : “Jill”, managerPath : [ “Mary”, “Jill” ] }
```

I would choose capturing the full manager path using arrays over just storing the immediate manager when the requirements state the need to do analysis across an employee’s complete management chain, such as answering the question, “What is the complete chain of command for Sven?” This trade-off is typical of the modeling choices the MongoDB modeler faces: run-time processing versus data redundancy.

Here is a collection showing the network example using parent references:



Employee :

```
{ _id : "Mary", employeeFirstName : "Mary" },
{ _id : "Bob", employeeFirstName : "Bob", managerID : [ "Mary", "Jill" ] },
{ _id : "Jill", employeeFirstName : "Jill", managerID : [ "Mary" ] },
{ _id : "Ken", employeeFirstName : "Ken", managerID : [ "Bob" ] },
{ _id : "Jane", employeeFirstName : "Jane", managerID : [ "Bob", "Ken", "Sven" ] },
{ _id : "Sven", employeeFirstName : "Sven", managerID : [ "Jill" ] }
```

Note that in these examples we are illustrating references, but if we would like to illustrate embedding the manager information with each employee, these collections would look very similar because we just have the two fields, **_id** and **employeeFirstName**. Also note that there are other embed and reference design options available in addition to what is shown in this example.

Keys

There is a lot of data out there, but how do you sift through it all to find what you’re looking for? That’s where keys come in. A key is one or more attributes whose purposes include enforcing rules, efficiently retrieving data and allowing navigation from one entity to another. This section explains candidate (primary and alternate),

surrogate, foreign, and secondary keys.

CANDIDATE KEY (PRIMARY AND ALTERNATE KEYS)

A candidate key is one or more data elements that uniquely identify an entity instance. The Library of Congress assigns an ISBN (International Standard Book Number) to every title. The ISBN uniquely identifies each title and is, therefore, the title's candidate key. When the ISBN for this title, *9781935504702*, is entered into many search engines and database systems, the book entity instance *Data Modeling for MongoDB* will be returned (try it!). **Tax ID** can be a candidate key for an organization. **Account Code** can be a candidate key for an account.

Sometimes a single attribute identifies an entity instance such as an **ISBN** for a title. Sometimes it takes more than one attribute to uniquely identify an entity instance. For example, both a **Promotion Type Code** and **Promotion Start Date** may be necessary to identify a promotion. When more than one attribute makes up a key, we use the term *composite key*. Therefore, **Promotion Type Code** and **Promotion Start Date** together are a composite candidate key for a promotion. A candidate key has three main characteristics:

- **Unique**. There cannot be duplicate values in the data in a candidate key, and it cannot be empty (also known as *nullable*). Therefore, the number of distinct values of a candidate key must be equal to the number of distinct entity instances. When the entity **Title** has **ISBN** as its candidate key, if there are 500 title instances, there will also be 500 unique ISBNs.
- **Stable**. A candidate key value on an entity instance should never change. If a candidate key value changes, it creates data quality issues because there is no way to determine whether a change is an update to an existing instance or a new instance.
- **Minimal**. A candidate key should contain only those attributes that are needed to uniquely identify an entity instance. If four data elements are listed as the composite candidate key for an entity but only three are needed for uniqueness, then only those three should be in the key.

For example, each **Student** may attend one or many **Classes**, and each **Class** may contain one or many **Students**. Here are some sample instances for each of these entities:

Student

Student Number	First Name	Last Name	Birth Date
SM385932	Steve	Martin	1/25/1958
EM584926	Eddie	Murphy	3/15/1971
HW742615	Henry	Winkler	2/14/1984
MM481526	Mickey	Mouse	5/10/1982
DD857111	Donald	Duck	5/10/1982
MM573483	Minnie	Mouse	4/1/1986
LR731511	Lone	Ranger	10/21/1949
EM876253	Eddie	Murphy	7/1/1992

Attendance

Attendance Date
5/10/2013

6/10/2013

7/10/2013

Class

Class Full Name	Class Short Name	Class Description Text
Data Modeling Fundamentals	Data Modeling 101	An introductory class covering basic data modeling concepts and principles.
Advanced Data Modeling	Data Modeling 301	A fast-paced class covering techniques such as advanced normalization and ragged hierarchies.
Tennis Basics	Tennis One	For those new to the game of tennis; learn the key aspects of the game.
Juggling		Learn how to keep three balls in the air at once!

Based on our definition of a candidate key (and a candidate key's characteristics of being unique, stable, and minimal) what would you choose as the candidate keys for each of these entities?

For **Student**, **Student Number** appears to be a valid candidate key. There are eight students and eight distinct values for **Student Number**. So unlike **Student First Name** and **Student Last Name**, which can contain duplicates like *Eddie Murphy*, **Student Number** appears to be unique. **Student Birth Date** can also contain duplicates, such as *5/10/1982*, which is the **Student Birth Date** for both *Mickey Mouse* and *Donald Duck*. However, the combination of **Student First Name**, **Student Last Name**, and **Student Birth Date** may make a valid candidate key.

For **Attendance**, we are currently missing a candidate key. Although the **Attendance Date** is unique in our sample data, we will probably need to know which student attended which class on this particular date.

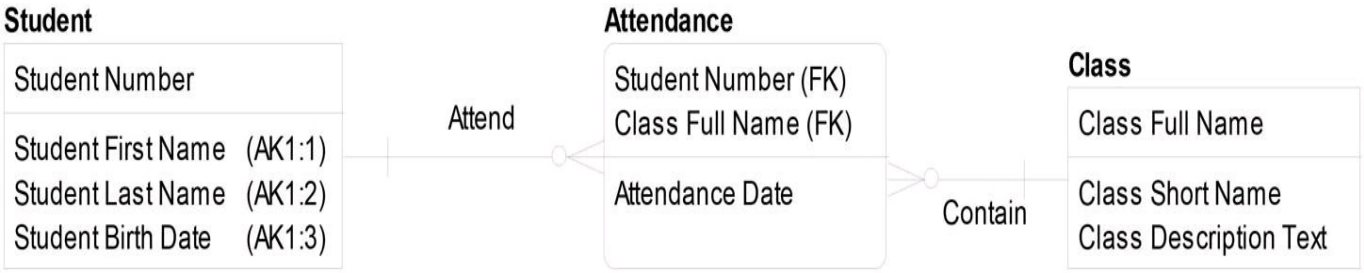
For **Class**, on first glance it appears that all of its attributes are unique and any of them would therefore qualify as a candidate key. However, *Juggling* does not have a **Class Short Name**. Therefore, because **Class Short Name** can be empty, we cannot consider it a candidate key. In addition, one of the characteristics of a candidate key is that it is stable. I know, based on my teaching experience, that class descriptions can change. Therefore, **Class Description Text** also needs to be ruled out as a candidate key, leaving **Class Full Name** as the best option for a candidate key.

Even though an entity may contain more than one candidate key, we can only select one candidate key to be the primary key for an entity. A primary key is the candidate key that has been chosen to be *the* unique identifier for an entity. An alternate key is a candidate key that, although it has the properties of being unique, stable, and minimal, was not chosen as the primary key but still can be used to find specific entity instances.

We have only one candidate key in the **Class** entity, so **Class Full Name** becomes our primary key. We have to make a choice in **Student**, however, because we have two candidate keys. Which **Student** candidate key would you choose as the primary key?

In selecting one candidate key over another as the primary key, consider succinctness and security. Succinctness means if there are several candidate keys, choose the one with the fewest attributes or shortest in length. In terms of privacy, it is possible that one or more attributes within a candidate key will contain sensitive data whose viewing should be restricted. We want to avoid having sensitive data in our entity's primary key because the primary key can propagate as a foreign key and therefore spread this sensitive data throughout our database. Foreign keys will be discussed shortly.

Considering succinctness and security in our example, I would choose **Student Number** over the composite **Student First Name**, **Student Last Name**, and **Student Birth Date**. It is more succinct and contains less sensitive data. Here is our data model with primary and alternate keys:



Primary key attributes are shown above the line in the rectangles. You will notice two numbers following the key abbreviation "AK." The first number is the grouping number for an alternate key, and the second number is the ordering of the attribute within the alternate key. So there are three attributes required for the **Student** alternate key: **Student First Name**, **Student Last Name**, and **Student Birth Date**. This is also the order in which the alternate key index will be created because **Student First Name** has a "1" after the colon, **Student Last Name** a "2," and **Student Birth Date** a "3."

Attendance now has as its primary key **Student Number** and **Class Full Name**, which appear to make a valid primary key. Note that the two primary key attributes of **Attendance** are followed by "FK". These are foreign keys, to be discussed shortly.

So to summarize, a candidate key consists of one or more attributes that uniquely identify an entity instance. The candidate key that is determined to be the best way to identify each record in the entity becomes the primary key. The other candidate keys become alternate keys. Keys containing more than one attribute are known as composite keys.

At the physical level, a candidate key is often translated into a unique index.

MongoDB Unique Index = RDBMS Unique Index (Candidate Key at Physical Level)

The concept of a candidate key in relational databases is equivalent to the concept of a unique index in MongoDB. If you want to make sure no two documents can have the same value in an ISBN, you can create a unique index:

db.title.ensureIndex({ ISBN : 1 }, { "unique" : true })

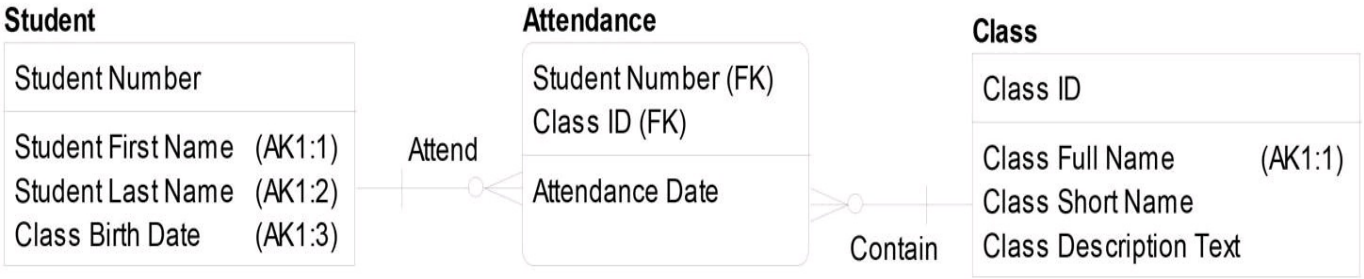
The *1* indicates that **ISBN** will be indexed in ascending order (as opposed to *-1*, which means descending order). The *true* means that **ISBN** must be unique. Note that to create a non-unique index, which will be discussed shortly, we would set the value to *false*. So if the above statement is executed and we try to create three documents that have the same **ISBN**, only the first occurrence would be stored and the next two occurrences would be skipped.

SURROGATE KEY

A surrogate key is a unique identifier for a row in a table, often a counter, that is always system-generated without intelligence, meaning a surrogate key contains values whose meanings are unrelated to the entities they identify. (In other words, you can't look at a month identifier of *1* and assume that it represents the **Month** entity instance value of *January*.) Surrogate keys should not be exposed to the business on user interfaces. They remain behind the scenes to help maintain uniqueness, to allow for more efficient navigation across structures, and to facilitate integration across applications.

You've seen that a primary key may be composed of one or more attributes of the entity. A single-attribute surrogate key is more efficient to use than having to specify three or four (or five or six) attributes to locate the single record you're looking for. Surrogate keys are useful for data integration, which is an effort to create a single, consistent version of the data. Applications such as data warehouses often house data from more than one application or system. Surrogate keys enable us to bring together information about the same entity instance that is identified differently in each source system.

When using a surrogate key, always make an effort to determine the natural key, which is what the business would consider to be the way to uniquely identify the entity, and then define an alternate key on this natural key. For example, assuming a surrogate key is a more efficient primary key than **Class Full Name**, we can create the surrogate key **Class ID** for **Class** and define an alternate key on **Class Full Name**. Note that the primary key column in **Attendance** also changes when we change the primary key of the related entity **Class**.



Class ID	Class Full Name	Class Short Name	Class Description Text
1	Data Modeling Fundamentals	Data Modeling 101	An introductory class covering basic data modeling concepts and principles.

2	Advanced Data Modeling	Data Modeling 301	A fast-paced class covering techniques such as advanced normalization and ragged hierarchies.
3	Tennis Basics	Tennis One	For those new to the game of tennis; learn the key aspects of the game.
4	Juggling		Learn how to keep three balls in the air at once!

Surrogate keys are frequently counters – that is, the first instance created is assigned the unique value of *1*, the second *2*, etc. However, some surrogate keys are Globally Unique Identifiers, or GUIDs for short. A GUID is a long and frequently randomly assigned unique value, an example being *21EC2020-3AEA-1069-A2DD-08002B30309D*.

MongoDB ObjectId = RDBMS GUID (Surrogate Key at Physical Level)

Every MongoDB document is identified by a unique **_id** field. Although the **_id** defaults to an ObjectId type, you may use another data type if you need to as long as it is unique within a collection. For example, you can have Collection *A* contain a document with an _id key of *12345* and Collection *B* contain a document with an _id key of *12345*, but Collection *A* cannot contain two documents with the _id key *12345*.

The ObjectId type is similar to a Globally Unique Identifier (GUID) in how values are often assigned, and therefore the chance of getting a duplicate value is very rare. The ObjectId is 12 bytes long. The first four bytes capture the timestamp down to the second, the next three bytes represent the host machine (whose value is usually hashed), and the next two bytes come from the process identifier of the ObjectId-generating process. These first nine bytes of an ObjectId guarantee its uniqueness within a single second. The last three bytes are an incrementing counter that is responsible for uniqueness within a second in a single process. This allows for up to 16,777,216 unique ObjectIds to be generated per process per second!

Here are some sample ObjectIds:

- ObjectId(“5280bd3711b3c2b87b4f555d”)
- ObjectId(“5280bd3711b3c2b87b4f555e”)

- ObjectId("5280bd3711b3c2b87b4f555f")

FOREIGN KEY

The entity on the "one" side of the relationship is called the parent entity, and the entity on the "many" side of the relationship is called the child entity. When we create a relationship from a parent entity to a child entity, the primary key of the parent is copied as a foreign key to the child.

A foreign key is one or more attributes that provide a link to another entity. A foreign key allows a relational database management system to navigate from one table to another. For example, if we need to know the customer who owns an account, we would want to include the **Customer ID** in the **Account** entity. The **Customer ID** in **Account** is the primary key for **Customer**.

Using this foreign key back to **Customer** enables the database management system to navigate from a particular account or accounts to the customer or customers that own each account. Likewise, the database can navigate from a particular customer or customers to find all of their accounts. Our data modeling tools automatically create a foreign key when a relationship is defined between two entities.

In our **Student**/**Class** model, there are two foreign keys in **Attendance**. The **Student Number** foreign key points back to a particular student in the **Student** entity, and the **Class ID** foreign key points back to a particular **Class** in the **Class** entity:

Student Number	Class ID	Attendance Date
SM385932	1	5/10/2013
EM584926	1	5/10/2013
EM584926	2	6/10/2013
MM481526	2	6/10/2013

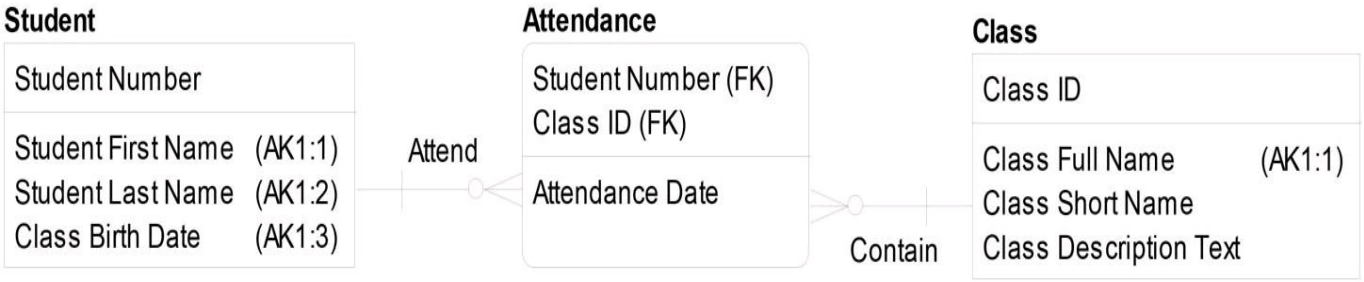
MM573483	2	6/10/2013
LR731511	3	7/10/2013

By looking at these values and recalling the students referenced by the **Student Number** and classes reference by **Class ID**, we learn that *Steve Martin* and *Eddie Murphy* both attended the *Data Modeling Fundamentals* class on *5/10/2013*. *Eddie Murphy* also attended the *Advanced Data Modeling Class* with *Mickey* and *Minnie Mouse* on *6/10/2013*. *Lone Ranger* took *Tennis Basics* (by himself, as usual) on *7/10/2013*.

MongoDB Reference = RDBMS Foreign Key

The concept of a foreign key in relational databases is similar to the concept of a reference in MongoDB. Both a foreign key and a reference provide a way to navigate to another structure. The difference, though, is that the relational database automatically ensures each foreign key value also exists as a value in the originating primary key. That is, if there is a **Product ID** foreign key in **Order Line** originating from **Product**, every **Product ID** value in **Order Line** must also exist in **Product**. This check of valid values from foreign keys back to their primary keys is called "referential integrity" (or RI for short). In MongoDB, the reference is simply a way to navigate to another structure (without the RI).

For example, recall our data model:



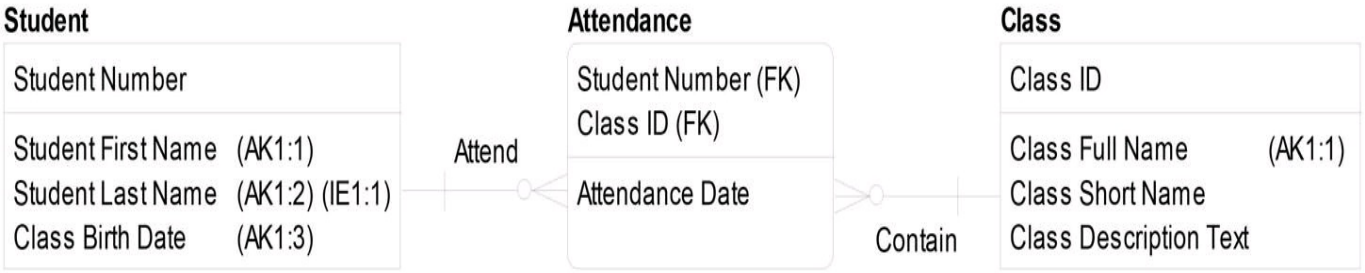
Student Number and **Class ID** in **Attendance** translate into foreign keys on the **Attendance** RDBMS table because of the constraints from **Student** to **Attendance** and from **Class** to **Attendance**. This means that a **Student Number** cannot exist in **Attendance** without having this **Student Number** first exist in **Student**. A **Class ID** cannot exist in **Attendance** without having this **Class ID** exist in **Class**.

However, in MongoDB, if **Attendance** is a collection with a **Student Number** reference to **Student** and a **Class** reference to **Class**, there are no checks to ensure the

Attendance Student Numbers and **Class IDs** point back to valid values in the **Student** and **Class** collections.

SECONDARY KEY

Sometimes there is a need to retrieve data rapidly from a table to answer a business query or meet a certain response time. A secondary key is one or more data elements (if there is more than one data element, it is called a composite secondary key) that are accessed frequently and need to be retrieved quickly. A secondary key is also known as a non-unique index or inversion entry (IE for short). A secondary key does not have to be unique, stable, nor always contain a value. For example, we can add an IE to **Student Last Name** in **Student** to allow for quick retrieval whenever any queries require **Student Last Name**:



Student Last Name is not unique, as there can be two *Jones*; it is not stable and can change over time; and sometimes we may not know someone's last name, so it can be empty.

MongoDB Non-Unique Index = RDBMS Secondary Key

The concept of a secondary key in relational databases is equivalent to the concept of a non-unique index in MongoDB. Indexes are added to improve retrieval performance. There are always tradeoffs with indexes, though. Balance retrieval speed with space (each index requires at least 8 kilobytes) and volatility (if the underlying data changes, the index has to change, too).

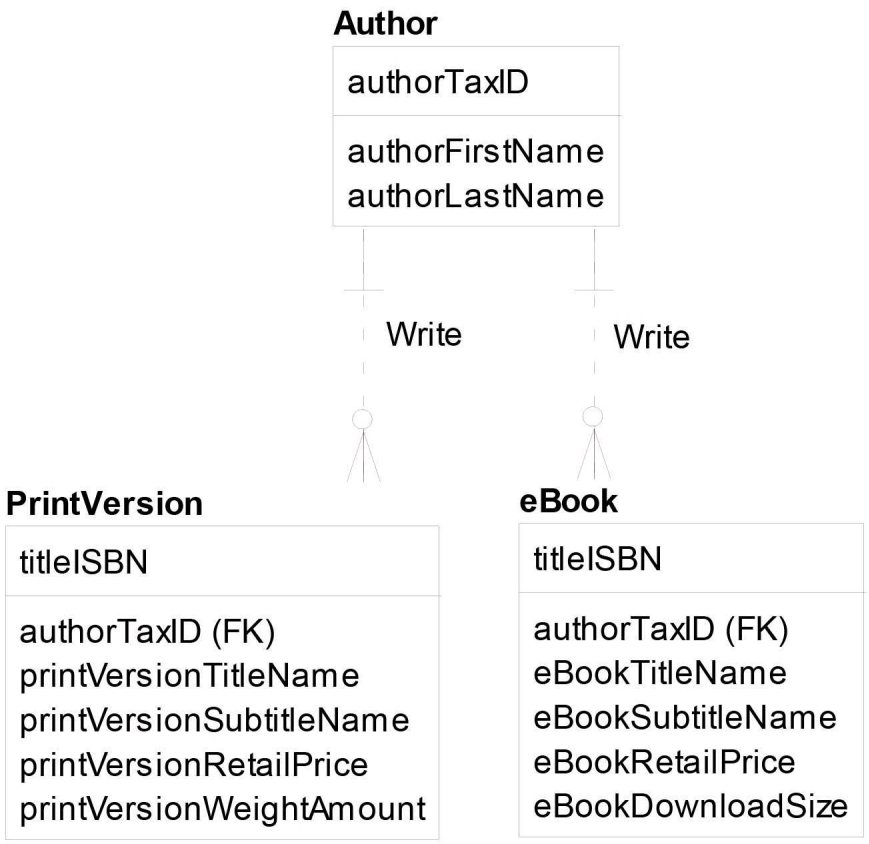
A query that does not use an index is called a "table scan" in an RDBMS and a "collection scan" in MongoDB. This means that the server has to look through all of the data, starting at the beginning, until all of the query results are found. This process is basically what you'd do if you were looking for information in a book without an index: you'd start at page one and read through the whole thing. In general, you want to avoid making the server do table scans because they is very slow for large collections.

SUBTYPE

Subtyping groups the common attributes and relationships of entities, while retaining what is special within each entity. Subtyping is an excellent way of communicating

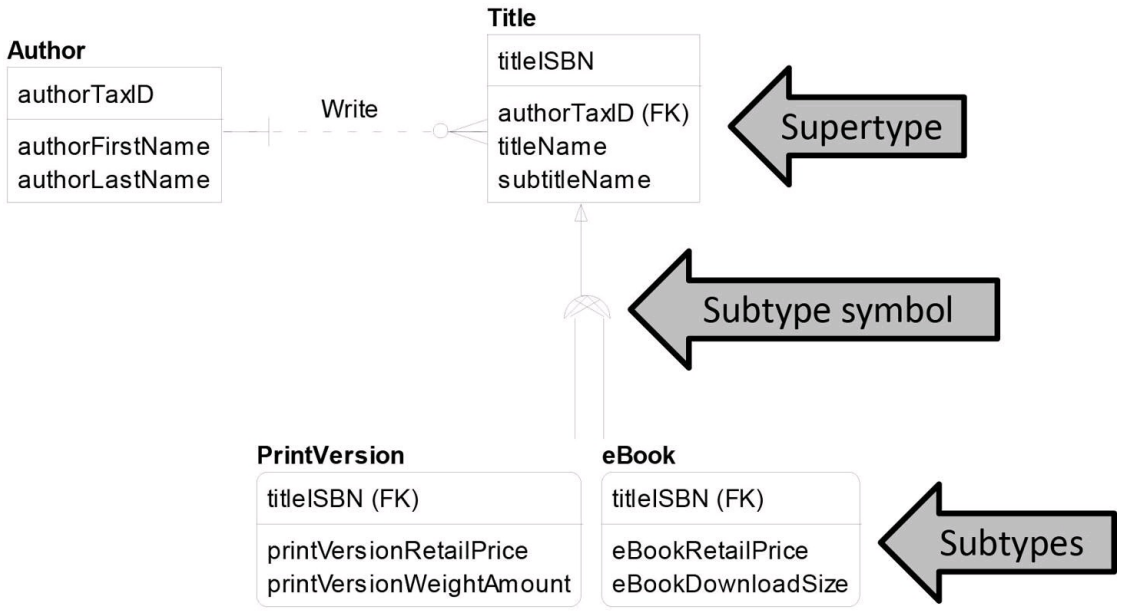
that certain concepts are very similar and of showing examples.

In our publishing data model, an **Author** may write many **PrintVersions** and many **eBooks**:



- Each **Author** may write one or many **PrintVersions**.
- Each **PrintVersion** must be written by one **Author**.
- Each **Author** may write one or many **eBooks**.
- Each **eBook** must be written by one **Author**.

Rather than repeat the relationship to **Author**, as well as the common attributes, we can introduce subtyping:



- Each **Author** may write one or many **Titles**.
- Each **Title** must be written by one **Author**.
- Each **Title** may be either a **PrintVersion** or **eBook**.
- Each **PrintVersion** is a **Title**.
- Each **eBook** is a **Title**.

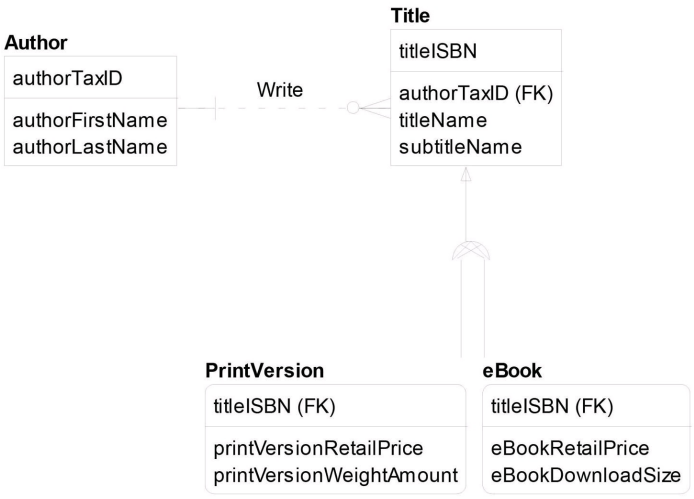
The subtyping relationship implies that all of the relationships and attributes from the supertype are inherited by each subtype. Therefore, there is an implied relationship from **Author** to **eBook** as well as from **Author** to **PrintVersion**. Also, the **titleName**, **subtitleName**, and **titleRetailPrice** belong to **PrintVersion** and belong to **eBook**. Note that each subtype must have the same attributes in their primary key as the supertype; in this case, **titleISBN**, which is the identifier for a particular title.

Not only does subtyping reduce redundancy on a data model, but it makes it easier to communicate similarities across what otherwise would appear to be distinct and separate concepts.

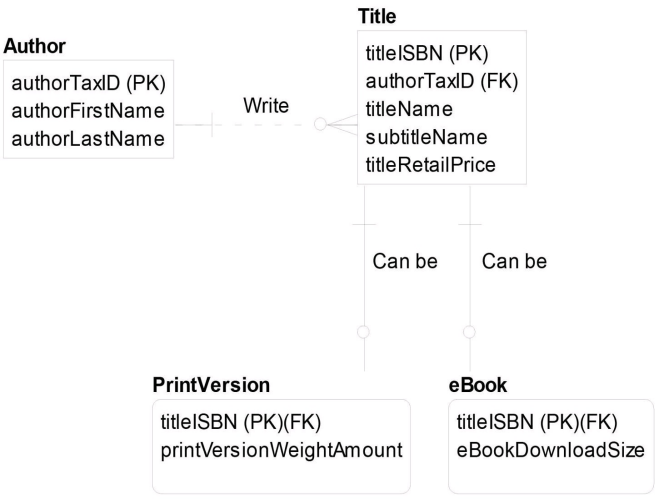
MongoDB Identity, Rolldown, Rollup = RDBMS Identity, Rolldown, Rollup (Subtype at Physical Level)

The subtyping symbol cannot exist on a physical data model, and the three ways to resolve are Identity, Rolldown, and Rollup:

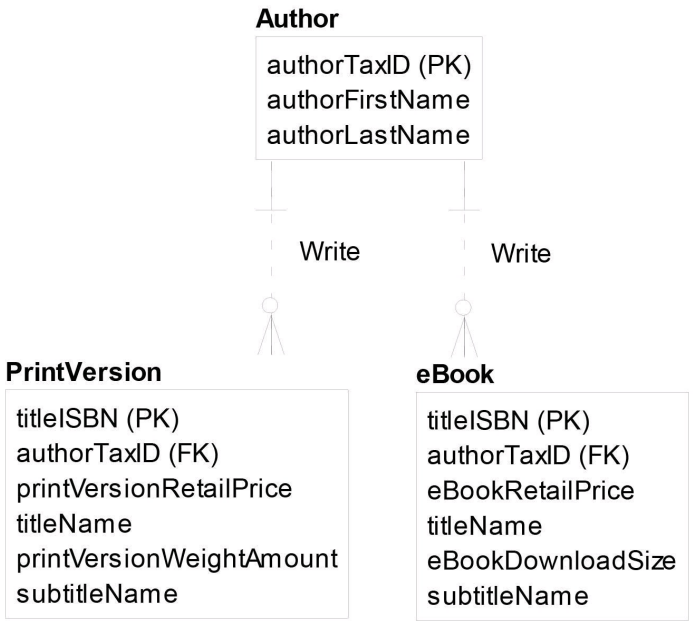
Original Logical



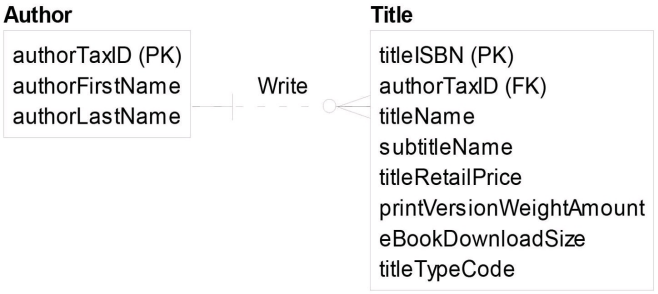
Identity



Rolldown



Rollup



Identity, rolldown, and rollup are the same techniques for both an RDBMS and in MongoDB:

- **Identity**. Identity is the closest to subtyping, because the subtyping symbol is replaced with a one-to-one relationship for each supertype/subtype combination.

The main advantage of identity is that all of the business rules at the supertype level and at the subtype level remain the same as in the logical data model. That is, we can continue to enforce relationships at the supertype or subtype levels as well as enforce that certain fields be required at the supertype or subtype levels. Identity allows us to still require the **printVersionWeightAmount** for print versions and the **eBookDownloadSize** for eBooks, for example. The main disadvantage of identity is that it can take more time to retrieve data as it requires navigating multiple tables to access both the supertype and subtype information.

- **Rolldown**. Rolldown means we are moving the attributes and relationships of the supertype down to each of the subtypes. Rolling down can produce a more user-friendly structure than identity or rolling up because subtypes are often more concrete concepts than supertypes, making it easier for the users of the data model to relate to the subtypes. However, we are repeating relationships and fields, which could reduce any user-friendliness gained from removing the supertype. In addition, the rolling down technique enforces only those rules present in the subtypes. This could lead to a less flexible data model as we can no longer easily accommodate new subtypes without modifying the data model. If a new type of **Title** is required in addition to **PrintVersion** and **eBook**, this would require effort to accommodate.

- **Rollup**. Rollup means rolling up the subtypes up into the supertype. The subtypes disappear, and all attributes and relationships only exist at the supertype level. Rolling up adds flexibility to the data model because new types of the supertype can be added, often with no model changes. However, rolling up can also produce a more obscure model as the audience for the model may not relate to the supertype as well as they would to the subtypes. In addition, we can only enforce business rules at the supertype level and not at the subtype level. For example, now **printVersionWeightAmount** and the **eBookDownloadSize** are optional instead of required. When we roll up, we often need a way to distinguish the original subtypes from each other, so we frequently add a type column such as **titleTypeCode**.

EXERCISE 2: Subtyping in MongoDB

Translate the Author / Title subtyping example we just reviewed into three MongoDB documents (rolldown, rollup, and identity) using the sample data on the facing page. Refer to Appendix A for the answer.

Author

authorTaxID	authorFirstName	authorLastName
22-5555555	Steve	Hoberman

Title

titleISBN	authorTaxID	titleName	subtitleName	titleRetailPrice
9780977140060	22-5555555	Data Modeling Made Simple	A Practical Guide for Business and IT Professionals	$44.95
9781935504702	22-5555555	Data Modeling for MongoDB	Building Well-Designed and Supportable MongoDB Databases	$39.95
9781935504719	22-5555555	Data Modeling for MongoDB	Building Well-Designed and Supportable MongoDB Databases	$34.95
9781935504726	22-5555555	The Best Data Modeling Jokes		$9.95

Print Version

titleISBN	printVersionWeightAmount
9780977140060	1.5
9781935504702	1

eBook

titleISBN	eBookDownloadSize
9781935504719	3
9781935504726	2

Key Points

- There are very strong connections between the data model, an RDBMS, and MongoDB:

Data Model	RDBMS	MongoDB
Entity instance	Record or Row	Document
Entity	Table	Collection
Attribute or Data element	Data element, Field or Column	Field
Attribute value or Data element value	Data element value, Field value, or Column value	Field value
Format domain	Datatype	Datatype

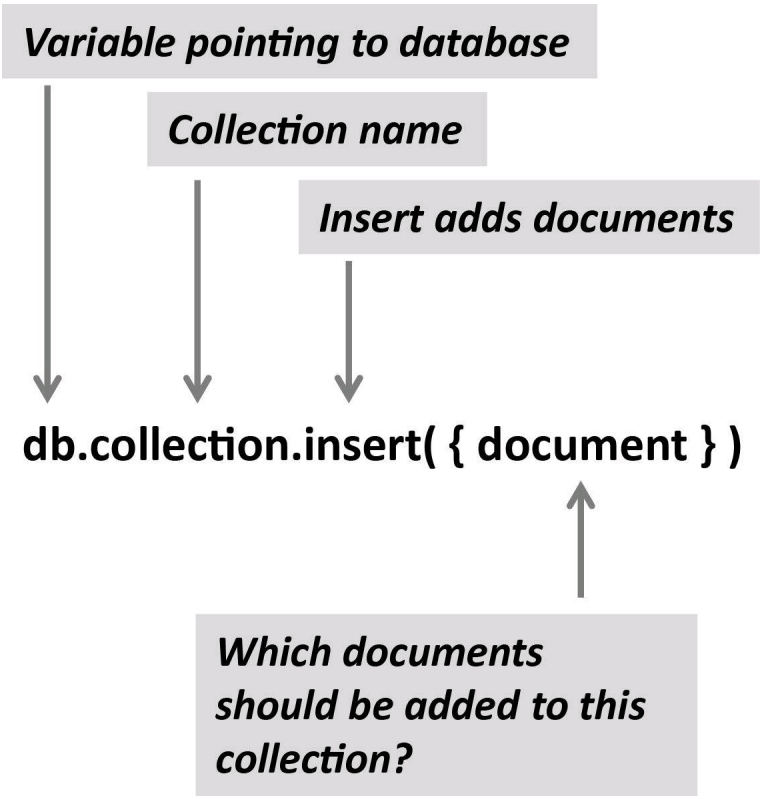
Relationship	Constraint	Captured but not enforced either through a reference, which is similar to a foreign key, or through embedded documents.
Candidate key (Primary or Alternate key)	Unique index	Unique index
Surrogate key	Globally Unique Id (GUID)	ObjectId
Foreign key	Foreign key	Reference
Secondary key, Inversion entry	Secondary key	Non-unique index
Subtype	Rolldown, Rollup, Identity	Rolldown, Rollup, Identity

Chapter 4
MongoDB Functionality

Now that you know the basic set of MongoDB objects, let's discuss what you can do with them through MongoDB functionality. We'll provide an overview to the four most common functions (adding, querying, updating, and deleting) as well as related operators. Because this is a book on how to do data modeling for MongoDB databases, and not a book on how to use MongoDB, we cover core functionality, but leave it to the MongoDB user guide and other resources on the market to cover the more advanced functionality (see Appendix B for a full list of MongoDB references).

Adding Data in MongoDB

The insert( ) function adds a document or an array of documents to a collection. The general structure for an insert( ) statement is:



There are three parts to the insert( ) function:

- **Database**. We need to identify which database we would like to add documents to. When we first start the MongoDB client, there is a variable called *db* that is assigned to represent the database for the current connection. Variables are pointers to the actual data they are assigned. Think of *db* as the pointer to your database. By default, *db* points to the test database. To see this yourself, type *db* at the command prompt and hit <Enter>.
- **Collection**. We need to identify the collection within this database we would like to add documents to.
- **Documents**. We need to list the one or more documents that will be added to the collection. We can add one document, or we can add many documents using an array.

For example, to add a new book document into the **book** collection, we can execute this statement:

```
db.book.insert( { titleName : "Data Modeling for MongoDB" } )
```

If the **book** collection does not exist, it will be created when adding this first document. The insert statement will also lead to the creation of an ObjectId field to

identify this new document.

Multiple documents can be added to a collection with one command by using an array:

```
db.book.insert( [ { titleName : “Data Modeling Made Simple” },
{ titleName : “Extreme Scoping” },
{ titleName : “UML Database Modeling Workbook” }
] )
```

To see what we just created, use the *find* command, which we’ll cover in the next section:

```
db.book.find( )
```

And this is what is returned:

```
{ “_id” : ObjectId(“530f8be4d77a08823086017c”), “titleName” : “Data Modeling for MongoDB” }
{ “_id” : ObjectId(“530f8be4d77a08823086017d”), “titleName” : “Data Modeling Made Simple” }
{ “_id” : ObjectId(“530f8be4d77a08823086017e”), “titleName” : “Extreme Scoping” }
{ “_id” : ObjectId(“530f8be4d77a08823086017f”), “titleName” : “UML Database Modeling Workbook” }
```

Note the ObjectIds were automatically created for us. We could optionally specify values for each ObjectId field. ObjectId is a default that’s good for guaranteeing uniqueness. However, if there are already unique keys in your data, you can use these unique keys for ObjectId values and this will save you an extra index.

The previous example illustrates when a document just contains the ObjectID and the book’s title. But what if we have more fields such as the three books in this spreadsheet?

Title Name	Subtitle Name	Page Count	Categories
Extreme Scoping	An Agile Approach to Enterprise Data Warehousing and Business Intelligence	300	Agile, Business Intelligence
Business unIntelligence	Insight and Innovation beyond Analytics and Big Data	442	Data Warehouse, Business Intelligence

Here is the statement to add these three new titles to our book collection:

```
db.book.insert( [  { titleName : “Extreme Scoping”,
subtitleName : “An Agile Approach to Enterprise Data Warehousing and Business Intelligence”,
pageCount : 300,
categories : [ “Agile”, “Business Intelligence” ]
},
{
titleName : “Business unIntelligence”,
subtitleName : “Insight and Innovation beyond Analytics and Big Data”,
pageCount : 442,
categories : [ “Data Warehouse”, “Business Intelligence” ]
},
{
titleName : “Secrets of Analytical Leaders”,
pageCount : 268,
categories : [ “Analytics” ]
} ] )
```

Note that not all documents have to contain all fields since the third book does not have a subtitle. Also note how easy it is to add multiple values for a single field through an array as in adding the books’ categories. Let’s do another find( ) and see what comes back:

```
db.book.find( )
```

And this is what is returned:

```
{ “_id” : ObjectId(“530f8be4d77a08823086017c”), “titleName” : “Data Modeling for MongoDB” }
{ “_id” : ObjectId(“530f8be4d77a08823086017d”), “titleName” : “Data Modeling Made Simple” }
{ “_id” : ObjectId(“530f8be4d77a08823086017e”), “titleName” : “Extreme Scoping” }
{ “_id” : ObjectId(“530f8be4d77a08823086017f”), “titleName” : “UML Database Modeling Workbook” }
{ “_id” : ObjectId(“530f8be4d77a08823086017g”), “titleName” : “Extreme Scoping”, “subtitleName” : “An Agile Approach to Enterprise Data Warehousing and Business Intelligence”, “pageCount” : 300, “categories” : [ “Agile”, “Business Intelligence” ] }
{ “_id” : ObjectId(“530f8be4d77a08823086017h”), “titleName” : “Business unIntelligence”, “subtitleName” : “Insight and Innovation beyond Analytics and Big Data”, “pageCount” : 442, “categories” : [ “Data Warehouse”,
```

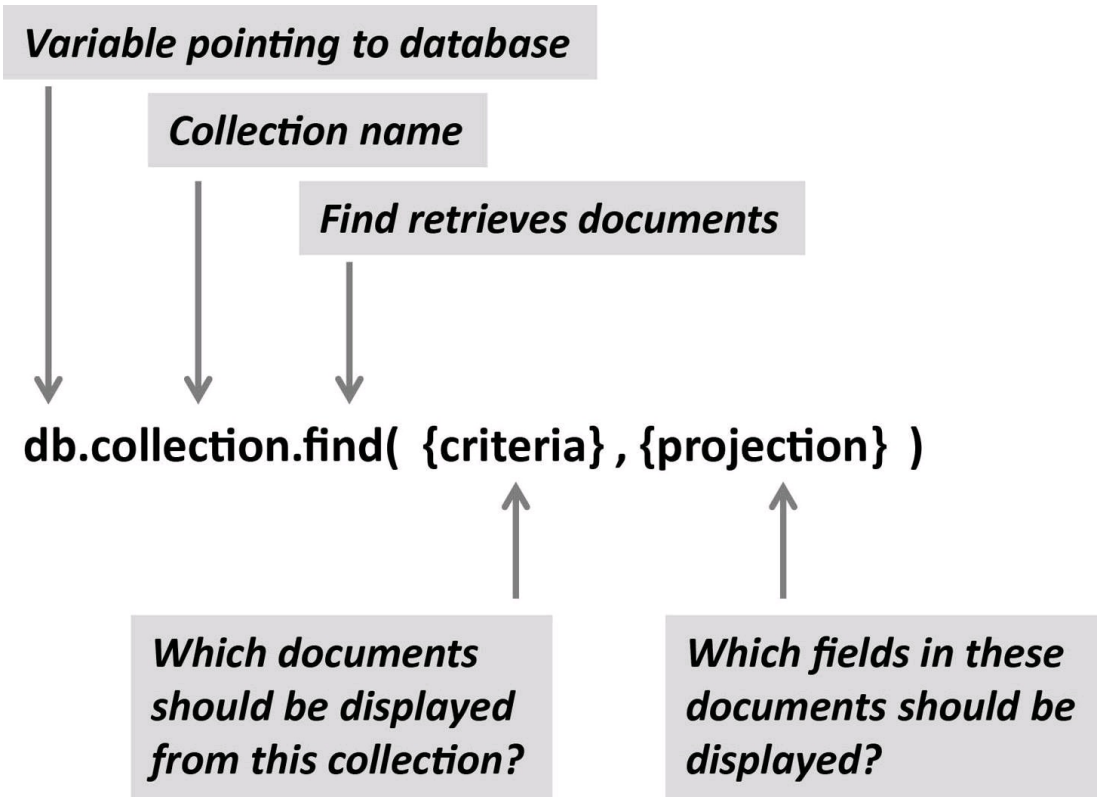
“Business Intelligence”] }

{ “_id” : ObjectId(“530f8be4d77a08823086017i”), “titleName” : “Secrets of Analytical Leaders”, “pageCount” : 268, “categories” : [“Analytics”] }

Notice that the *Extreme Scoping* book now exists twice in the collection. If this is indeed the same book we will need to remove the duplicate, which we will do shortly with remove( ).

Querying Data in MongoDB

The find( ) function displays documents from a collection. The general structure for a find( ) statement is:



There are three parts to the find( ) function:

- **Database**. We need to identify which database we are querying. More accurately, the variable pointing to the database we are querying, such as in this case *db*.

- **Collection**. We need to identify the collection within this database we are querying.

- **Parameters**. We need to identify the documents and fields within this collection we would like displayed. The first parameter contains the criteria for displaying the documents, including optionally using operators to bring back one or more documents. The second parameter called "projection" specifies which fields to display. In RDBMS terms, the first parameter specifies the rows and the second specifies the columns. Note that if we leave out both parameters, all documents will be displayed (up to the first 20 and then more can be displayed by typing it).

So if we would like to display all books in our book collection, we can execute:

```
db.book.find( )
```

If we would like to bring back a specific book in our collection, we can execute:

```
db.book.find( { titleName : "Data Modeling for MongoDB" } )
```

If we would like to display a specific book in our collection and only the **titleName** field (along with **_id**), we can execute:

```
db.book.find( { titleName : "Data Modeling for MongoDB" },
{ titleName : 1 } )
```

The *1* after a field name tells MongoDB we would like this field displayed. To exclude a field, use a *0*. So if we would like all of the fields displayed except the **titleName** field, we can execute:

```
db.book.find( { titleName : "Data Modeling for MongoDB" }, { titleName : 0 } )
```

If there is more than one field in the find clause, there is an implied "AND" between the conditions. That is, all conditions must be true. So, for example, to find the book titled *Extreme Scoping*, which contains *300* pages, we can write this query:

```
db.book.find( { titleName : "Extreme Scoping", pageCount : 300 } )
```

You can include any of the following operators as well as other operators described in the MongoDB reference manual:

- Greater than: $gt
- Great than or equal to: $gte
- Less than: $lt
- Less than or equal to: $lte
- Not equal to: $ne
- Match any or all conditions: $or
- Match all conditions: $and
- Match all conditions not specified: $not

- Match any of the values that exist in an array: $in
- Match all documents that contain a specified field: $exists
- Increment the value of a field by a certain amount: $inc

For example, if we would like to find all of the books in this list that are over 275 pages:

Extreme Scoping, 300 pages

Data Engineering, 100 pages

Business unIntelligence, 442 pages

Secrets of Analytical Leaders, 268 pages

We can run this query:

```
db.book.find( { pageCount : { $gt : 275 } } )
```

And this would be returned:

Extreme Scoping, 300 pages

Business unIntelligence, 442 pages

We can also add functions to the end of the find statement for further query refinement. If we wanted to sort the documents returned by **titleName**, we can run this query:

```
db.book.find( { pageCount : { $gt : 275 } } ).sort( {titleName : 1 } )
```

And this would be returned:

Business unIntelligence, 442 pages

Extreme Scoping, 300 pages

Note that the *1* in the sort command indicates ascending order and *-1* indicates descending order.

If we wanted to count the number of documents returned in this same find( ) query, we can execute this statement:

```
db.book.find( { pageCount : { $gt : 275 } } ).count( )
```

This statement would return *2* for the two records returned.

EXERCISE 3: INTERPRETING QUERIES

Based on the sample dataset below from the **Title** collection, what would each of the following two queries return? See Appendix A for the answer.

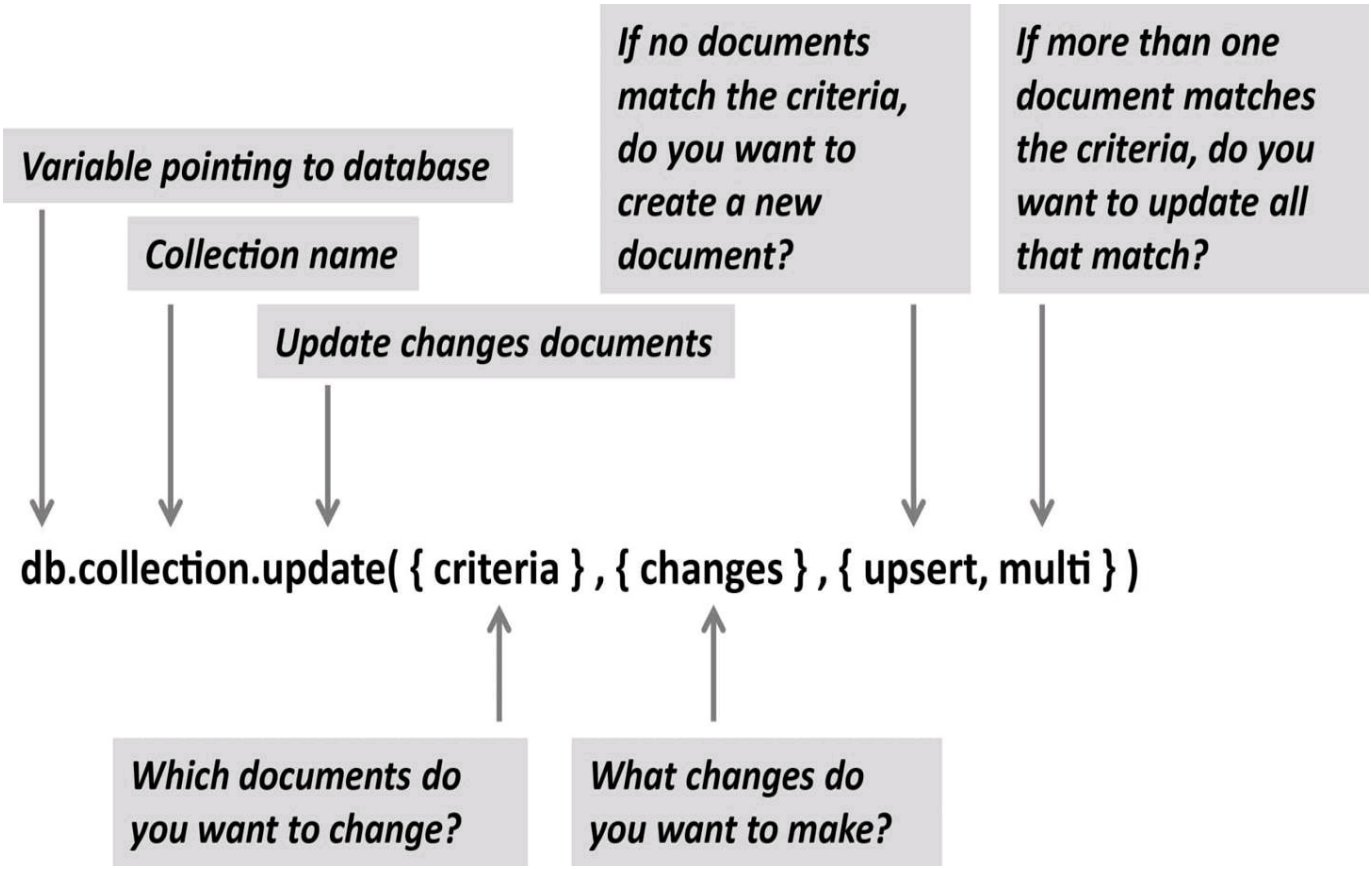
Title Name	Page Count	Publication Year	Author	Amazon Review
Extreme Scoping	300	2013	Larissa Moss	4.5
Data Engineering	100	2013	Brian Shive	4.25
Business unIntelligence	442	2013	Barry Devlin, PhD	5
Data Modeling Made Simple	250	2009	Steve Hoberman	4.35

```
db.title.find( { $or : [ { pageCount : { $gt : 200 } },
{ publicationYear : { $lt : 2010 } } ] } )
```

```
db.title.find( { authorName : { $ne : “Shive” } } )
```

Updating Data in MongoDB

The update( ) function modifies one or more documents from a collection. The general structure for a update( ) statement is:



There are three parts to the update( ) function:

- **Database**. We need to identify which database we are updating. More accurately, the variable pointing to the database we are updating, such as in this case *db*.
- **Collection**. We need to identify the collection within this database we are updating.
- **Parameters**. We need to identify the documents within this collection we would like changed. The first parameter contains the criteria for finding the documents we would like changed, which is identical to the syntax in the find( ) function because we are querying for documents. The second parameter, called "changes," specifies how we would like to change the documents we find. We may decide to modify the values in one or more fields. The third parameter contains two indicators that can appear in any order: *upsert* and *multi*. Upsert means "update else insert," meaning if this indicator is set to *true* or *1* and the criteria does not bring back any documents in the collection to update, then a new document will be created. If we leave out the upsert parameter or set it to *false* or *0*, a new document will not be created. Multi is an indicator that, if set to *true*

or *1*, tells MongoDB to update all of the documents that meet the criteria. If we leave out the multi parameter or set it to *false* or *0*, only the first document encountered will be updated.

So, if we would like to update the title for *Data Modeling for MongoDB*, we can execute this statement:

```
db.book.update( { titleName : "Data Modeling for MongoDB" },
{ titleName : "Fifty Shades of Data Modeling" } )
```

Because we left off the multi and upsert parameters, we would only change the first occurrence of this title, and if this title did not exist in our collection, we would not insert this title as a new document.

If we wanted to update all documents that meet this criteria, we can execute this statement:

```
db.book.update( { titleName : "Data Modeling for MongoDB" },
{ titleName : "Fifty Shades of Data Modeling" }, { multi: true } )
```

Note that the basic update function is designed for addressing the situation when what you are searching for and what you are changing are the same fields. If we wanted to change a different field than the one being searched for, you will need to use the $set command:

```
db.book.update ( {_id : ObjectId("530f8be4d77a08823086017d") },
{ "$set" : { titleName : "Fifty Shades of Data Modeling" } } )
```

What would happen if we wanted to update a field but the field did not exist? The solution is to again use $set. $set sets the value of a field. If the field does not yet exist, it will be created. If the field does exist, it will be updated with the value specified by $set. This can be handy for updating schema or adding user-defined fields. For example, if we wanted to update the edition but **edition** did not exist, we can run this statement:

```
db.book.update ( { titleName : "Fifty Shades of Data Modeling" },
{ "$set" : { edition : 2 } } )
```

Now this document will have an **edition** field:

```
db.book.find( { titleName : "Fifty Shades of Data Modeling" } )
{ "_id" : ObjectId("530f8be4d77a08823086017d"), "titleName" : "Fifty Shades of Data Modeling", "edition" : 2 }
```

If we decided that this book is really the third edition and not the second edition, $set can be used again to change the value:

```
db.book.update( { titleName : "Fifty Shades of Data Modeling" },
```

{ “$set” : { edition : 3 } })

$set can even change the type of the field it modifies. For instance, if we decide that we would like to store the character string *three* instead of *3*, we can run this statement:

db.book.update ({ titleName : “Fifty Shades of Data Modeling” },

{ “$set” : { edition : “three” } })

If we decide the **edition** field really isn’t needed anyway, we can remove this field altogether with $unset:

db.book.update ({ titleName : “Fifty Shades of Data Modeling” }, { “$unset” : { edition : “three” } })

The $inc modifier can be used to increment the value of an existing field and, similar to $set, can be used to create a new field if it does not already exist. For example, let’s say we would like to add a field called **bookTotalCopiesSoldCount**, which specifies how many copies of a title have been sold since the book was first published. We can run this statement:

db.book.update ({ titleName : “Fifty Shades of Data Modeling” },

{ “$inc” : { bookTotalCopiesSoldCount : 3000 } })

If we look at the document after this update, we’ll see the following:

db.book.find ({ titleName : “Fifty Shades of Data Modeling” })

{ “_id” : ObjectId(“530f8be4d77a08823086017d”), “titleName” : “Fifty Shades of Data Modeling”, “bookTotalCopiesSoldCount” : 3000 }

Next month, after 100 more copies of this title are sold, we can run this statement to add 100 more copies to the total:

db.book.update ({ titleName : “Fifty Shades of Data Modeling” },

{ “$inc” : { bookTotalCopiesSoldCount : 100 } })

If we look at the document after this update, we’ll see the following:

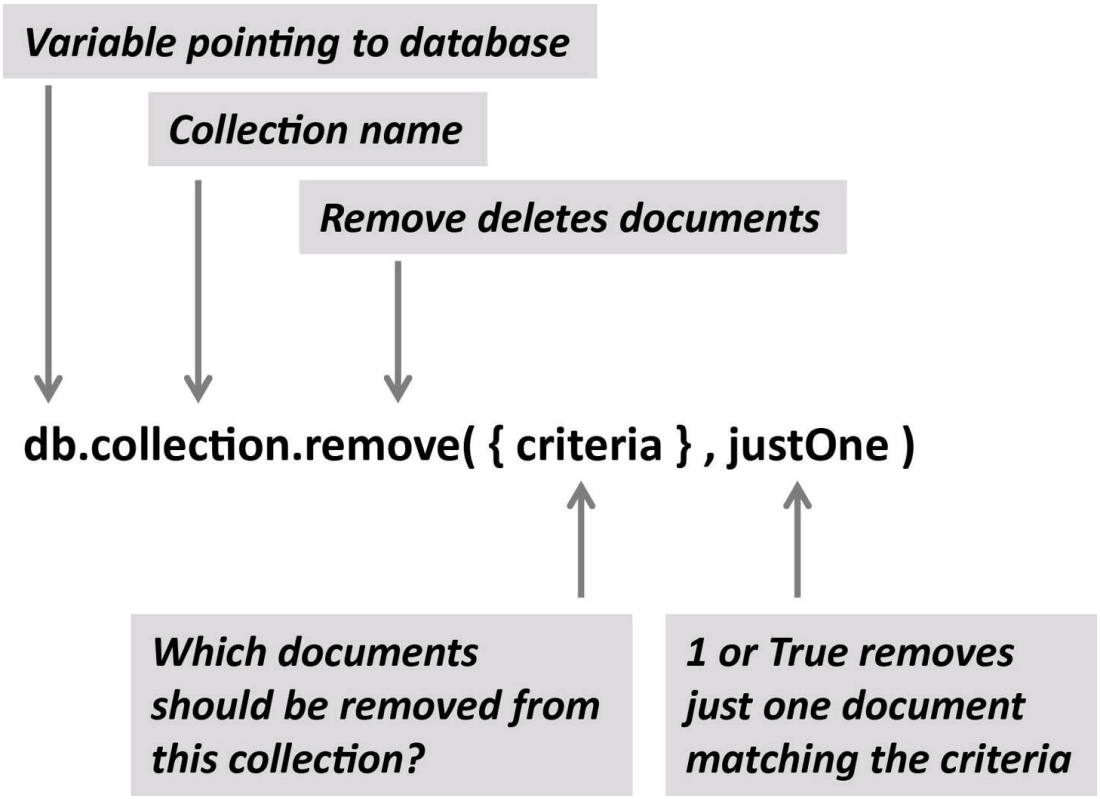
db.book.find({ titleName : “Fifty Shades of Data Modeling” })

{ “_id” : ObjectId(“530f8be4d77a08823086017d”), “titleName” : “Fifty Shades of Data Modeling”, “bookTotalCopiesSoldCount” : 3100 }

There are many more ways to update data and especially ways to update data in arrays. Please refer to the MongoDB user guide for more information.

Deleting Data in MongoDB

The remove( ) function deletes one or more documents from a collection. The general structure for a remove( ) statement is:



There are three parts to the remove( ) function:

- **Database**. We need to identify which database we are removing data from. More accurately, the variable pointing to the database, such as in this case *db*.
- **Collection**. We need to identify the collection we are removing data from.
- **Parameters**. We need to identify the criteria for the documents that need to be removed. Setting the justOne parameter to *true* or *1* removes just the first occurrence of the criteria. If we leave out the justOne parameter or set it to *false* or *0*, all documents that meet the criteria will be removed.

To remove all of the documents within the book collection, we can execute the remove( ) statement without any parameters:

```
db.book.remove( )
```

To remove a particular book, we can execute this statement:

```
db.book.remove( { titleName : “Fifty Shades of Data Modeling” } )
```

Note that if we had five books with the same title, the above statement will remove all

five of them.

Recall earlier that we added the book *Extreme Scoping* twice in our collection:

```
{ _id : ObjectId("530f8be4d77a08823086017e"), titleName : "Extreme Scoping" }

{ _id : ObjectId("530f8be4d77a08823086017g"), titleName : "Extreme Scoping", "subtitleName" : "An Agile Approach to Enterprise Data Warehousing and Business Intelligence", pageCount : 300, categories : [ "Agile", "Business Intelligence" ] }
```

We can now remove the duplicate:

```
db.book.remove( { titleName : "Extreme Scoping" } )
```

But don't hit the enter button just yet! This will remove both occurrences of *Extreme Scoping*, and we only want to remove the first one. Executing the following statement would remove just the first one:

```
db.book.remove( { _id : ObjectId("530f8be4d77a08823086017e") } )
```

Instead of running the above statement, we could also just remove the first occurrence of *Extreme Scoping* by running this statement:

```
db.book.remove( { titleName : "Extreme Scoping" }, 1 )
```

Using *1* or *true* as the second parameter in the remove statement deletes only the first occurrence. Using *0* or *false* or leaving this second parameter off completely deletes all of the documents that meet the specified criteria.

EXERCISE 4: MONGODB FUNCTIONALITY

Below is some sample data from the **Title** entity. Write the MongoDB statements to insert this data into the collection **Title**. Then remove *FruITion*. Then update the page count for *Data Quality Assessment* to *350*. Then view all of the data. See Appendix A for the answer.

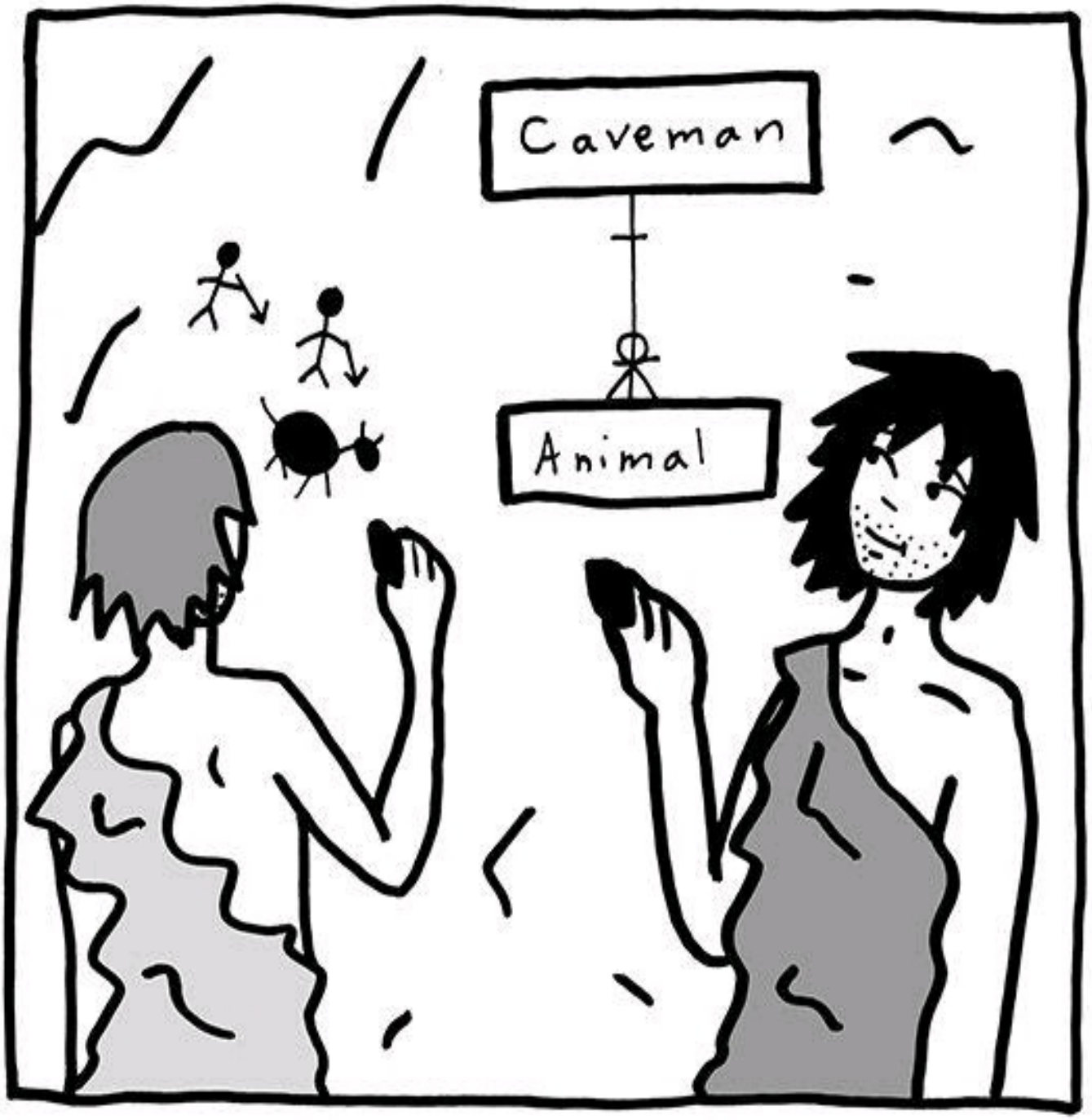
Title Name	Page Count	Publication Year	Author
FruITion	100	2010	Chris Potts
Data Quality Assessment	400	2009	Arkady Maydanchik

Key Points

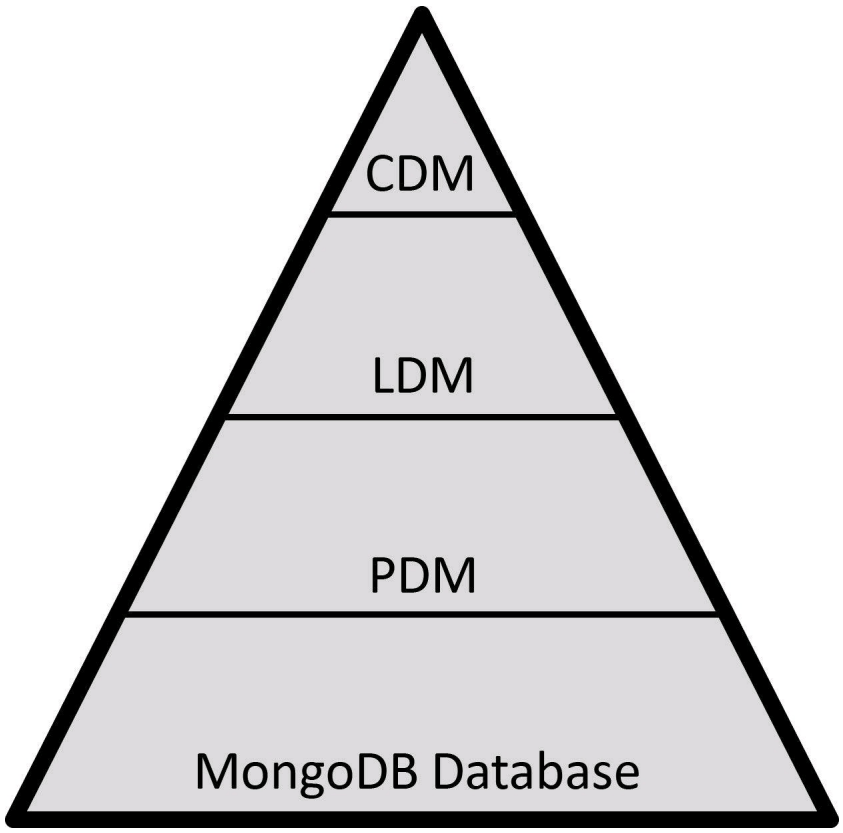
- To add one or more documents to a collection, use the insert( ) function.
- To retrieve one or more documents from a collection, use the find( ) function.
- To modify one or more documents in a collection, use the update( ) function.
- To delete one or more documents in a collection, use the delete( ) function.
- Although the basic functionality was covered in this chapter, there is a lot more to learn about MongoDB including more functions, more operators, and knowing the efficiency tradeoffs for each option.

Section II

Levels of Granularity

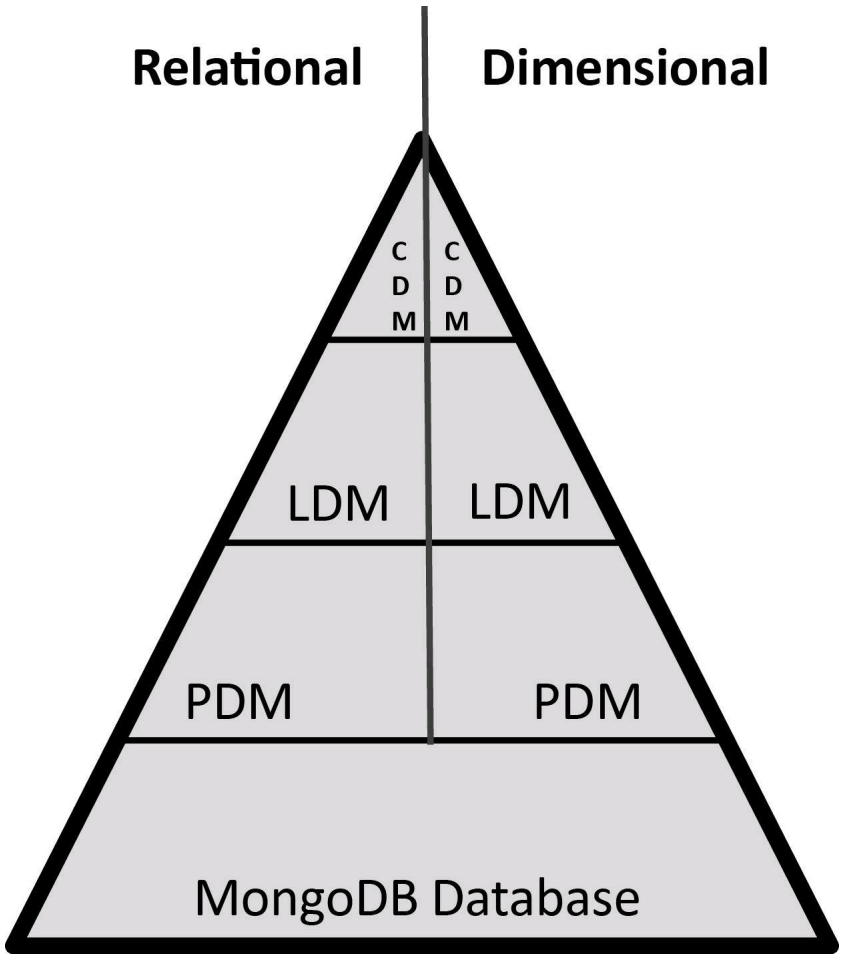


This section covers Conceptual Data Modeling (Chapter 5), Logical Data Modeling (Chapter 6), and Physical Data Modeling (Chapter 7). Notice the "ing" at the end of each of these chapter titles. We focus on the process of building each of these models, which is where we gain essential business knowledge. This pyramid summarizes the four levels of design:



At the highest level, we have the Conceptual Data Model (CDM), which captures the satellite view of the business solution. The CDM provides the context for and scope of the Logical Data Model (LDM), which provides the detailed business solution. The LDM becomes the starting point for applying the impact of technology in the Physical Data Model (PDM). The PDM represents the actual MongoDB database structures.

In addition to the conceptual, logical, and physical levels of detail, there are also two different modeling mindsets: relational and dimensional.



Relational data modeling is the process of capturing how the business *works* by precisely representing business rules, while dimensional data modeling is the process of capturing how the business is *monitored* by precisely representing business questions.

The major difference between relational and dimensional data models is in the meaning of the relationships. On a relational data model, a relationship communicates a business rule, while on a dimensional data model, the relationship communicates a navigation path. On a relational data model, for example, we can represent the business rule "A **Customer** must have at least one **Account**." On a dimensional data model, we can display the measure **Gross Sales Amount** and all of the navigation paths from which a user needs to see **Gross Sales Amount** such as by day, month, year, region, account, and customer. The dimensional data model is all about answering business questions by viewing measures at different levels of granularity.

The following table summarizes these three levels of design and two modeling mindsets, leading to six different types of models:

		Mindset	
		Relational	**Dimensional**
Levels of Design	**CDM**	Key concepts and their business rules such as, "Each **Customer** may place one or many **Orders**."	Key concepts focused on answering a set of business questions such as, "Can I see **Gross Sales Amount** by **Customer**?"
	LDM	All attributes required for a given application, neatly organized into entities according to strict business rules, and independent of technology such as, "Each **Customer ID** value must return, at most, one **Customer Last Name**."	All attributes required for a given analytical application, focused on answering a set of business questions and independent of technology, such as, "Can I see **Gross Sales Amount** by **Customer** and view the customer's first and last name?"
	PDM	The LDM modified to perform well in MongoDB. For example, "To improve retrieval speed, we need a non-unique index on **Customer Last Name**."	The LDM modified to perform well in MongoDB. For example, "Because there is a need to view **Gross Sales Amount** at a **Day** level, and then by **Month** and **Year**, we should consider embedding all calendar fields into a single collection."

Note that it seems like there is a lot of work to do; we need to go through all three phases – conceptual, logical, and physical. Wouldn't it be easier to just jump straight to building a MongoDB database and be done with it?

Going through the proper levels of design will take more time than just jumping into building a MongoDB database. However, the thought process we go through in

building the application should ideally cover the steps we go through during these three levels of design anyway. For example, if we jump straight into building a MongoDB database, we would still need to ask at least some of the questions about definitions and business rules; it's just that we would do it all at once instead of in separate phases. Also, if we don't follow these modeling steps proactively, we will be asking the questions during support, where fixing things can be much more expensive in terms of time, money, and reputation. Believe me, I know—many of my consulting assignments involve fixing situations due to skipping levels of design (e.g., jumping right to the physical). I can't tell you how many times during my assignments I have heard a manager use the phrase "technical debt" to summarize the high cost to maintain and poor performance of applications built without conceptual and logical data models. For example, take the MongoDB document we created in the previous chapter:

```
{
titleName : "Extreme Scoping",
subtitleName : "An Agile Approach to Enterprise Data Warehousing and Business Intelligence",
pageCount : 300
}
```

This is a very simple document with just three fields: the book's title name, subtitle name, and page count. However, even with just these three fields, there are conceptual, logical, and physical questions that need to be answered.

During conceptual data modeling, we would address questions such as these:

- What is the right name for the concept of "book"? Should we call it a "book" or a "title" or a "copy" or an "intellectual unit"?
- What is a clear, correct, and complete definition for the concept of "book"? Once we get this definition agreed upon and documented, nobody else will need to go through this painful definition process again.
- Is the scope only book, or can it include other important concepts such as author, publisher, and customer? That is, what is the scope of the application?
- Can a book exist without an author?
- Can a book be written by more than one author?

During logical data modeling, we would address questions such as these:

- Is the book's title name required? Is subtitle name required? Is page count required?
- Can a book have more than one subtitle?
- How do you identify a book?

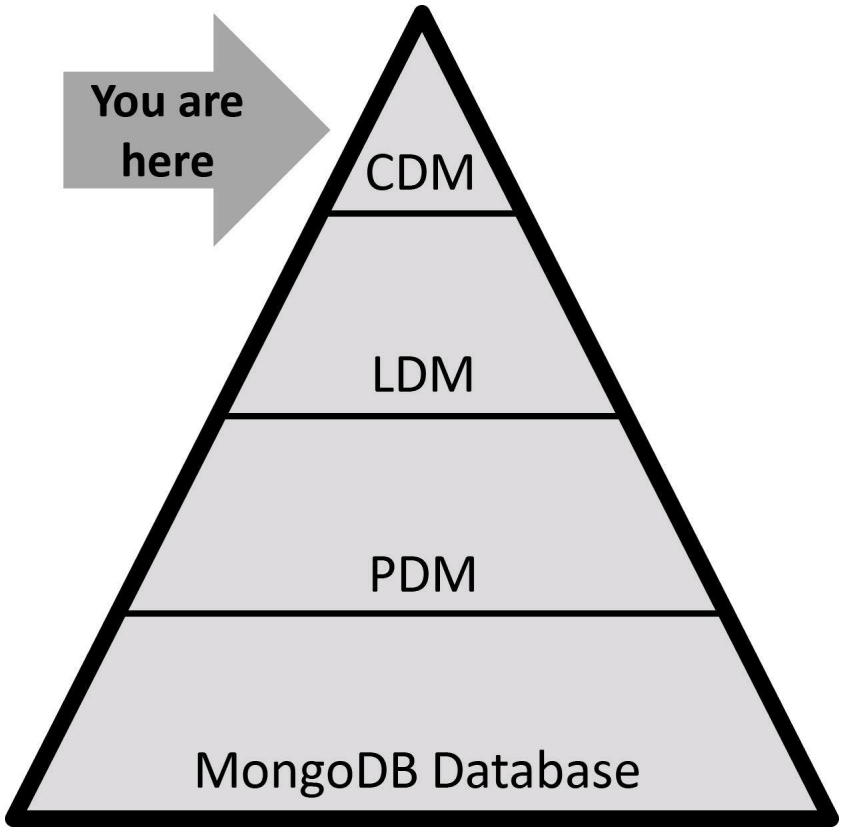
- Is an eBook considered a book?
- Does an eBook have a page count?

During physical data modeling, we would address questions such as these:

- How many books do we have, and therefore, how much space will we need?
- What are the performance impacts of loading and retrieving book data?
- Where do we need additional indexes to further improve retrieval performance?
- Should we embed or reference?
- What are the history requirements?

By the end of this section, you will be able to appreciate, understand, and complete the three different phases of modeling for MongoDB applications.

Chapter 5
Conceptual Data Modeling



Conceptual data modeling is the process of capturing the satellite view of the business requirements. What problems does the business need to solve? What do these ideas or concepts mean? How do these concepts relate to each other? What is the scope of the effort? These are types of questions that get addressed during conceptual data modeling.

The end result of the conceptual data modeling phase is a conceptual data model (CDM) that shows the key concepts needed for a particular application development

effort. This chapter defines a concept and offers an explanation of the importance of conceptual data modeling. We'll walk through a simple five-step, template-driven approach that works well with MongoDB.

Concept Explanation

A concept is a key idea that is both *basic* and *critical* to your audience. "Basic" means this term is probably mentioned many times a day in conversations with the people who are the audience for the model, which includes the people who need to validate the model as well as the people who need to use the model. "Critical" means the business would be very different or non-existent without this concept.

The majority of concepts are easy to identify and include those that are common across industries such as **Customer**, **Employee**, and **Product**. An airline may call a **Customer** a **Passenger**, and a hospital may call a **Customer** a **Patient**, but in general they are all people who receive goods or services. Each concept will be shown in much more detail at the logical and physical phases of design. For example, the **Customer** concept might encompass the logical entities **Customer**, **Customer Association**, **Customer Demographics**, **Customer Type**, and so on.

Many concepts, however, can be more challenging to identify as they may be concepts to your audience but not to others in the same department, company, or industry. For example, **Account** would most likely be a concept for a bank and for a manufacturing company. However, the audience for the bank conceptual data model might also require **Checking Account** and **Savings Account** to be within scope, whereas the audience for the manufacturing conceptual data model might, instead, require **General Ledger Account** and **Accounts Receivable Account** to be within scope.

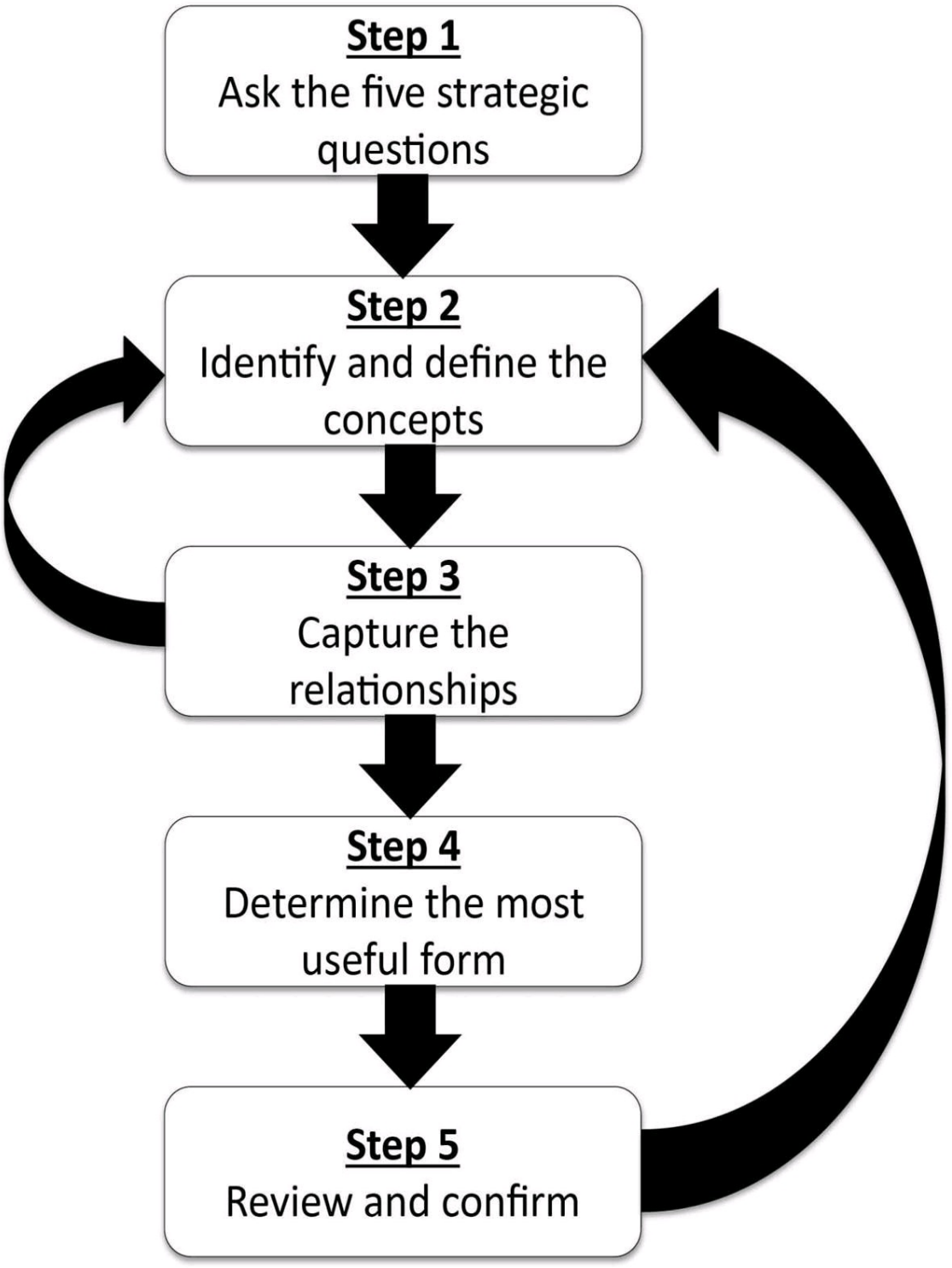
The main reason for conceptual data modeling is to get the "big picture" and understand the scope and high level needs of the project. Aligning on a common understanding at this level leads to these benefits:

- **Scope and direction determined**. We can capture extremely complex and encompassing application requirements on a single piece of paper. After capturing such a broad perspective, we can more easily identify a subset of the model to analyze. For example, we can model the entire Logistics department and then scope out of this a particular logistics application that we would like to build. The broad perspective of conceptual data modeling can help us determine how planned and existing applications will coexist. It can provide direction and guidance on what new functionality the business will need next.

- **Proactive analysis performed**. By developing a high-level understanding of the application, there is a strong chance we will be able to identify important issues or concerns proactively, saving substantial time and money later on. Examples include term definition differences and different interpretations of project scope. I once worked on a project with a university, and we didn't realize the term **Student** had multiple conflicting definitions until we started conceptual data modeling!
- **Key terms defined**. The conceptual data model contains the most important concepts of the project and requires that each of these concepts be defined and documented. Clear and complete definitions at this level are essential for effective communication across the project team. When well-defined terms are mentioned during meetings or in project documentation, people will be clear on what these terms mean or have the ability to find out because these terms are documented, ideally in an accessible and user-friendly format such as a wiki.

Conceptual Data Modeling Approach

Conceptual data modeling is the process of capturing the satellite view of the business requirements. There are five steps to conceptual data modeling, as illustrated in the diagram on the next page. Before you begin any project, there are five strategic questions that must be asked (Step 1). These questions are a prerequisite to the success of any application effort. Next, identify all of the concepts within the scope of the application you are building (Step 2). Make sure each concept is clearly and completely defined. Then determine how these concepts are related to each other (Step 3). Often, you will need to go back to Step 2 at this point because in capturing relationships you often come up with new concepts—and then there are new relationships between these concepts. Next, decide the most useful form for making sure your work during this phase is put to good use (Step 4). Someone will need to review your work and use your findings during development, so deciding on the most useful form is an important step. As a final step, review your work to get approval to move on to the logical data modeling phase (Step 5).



Frequently, action items come out of the review and confirm step, causing you to go back to Step 2 and refine your concepts. Expect iteration through these steps. Let's talk

more about each of these five steps.

STEP 1: ASK THE FIVE STRATEGIC QUESTIONS

Ask these five questions as early as possible in the application development process:

1. *What is the application going to do?* Precisely and clearly document the answer to this question in less than three sentences. Make sure to include whether you are replacing an existing system, delivering new functionality, integrating several existing applications together, etc. Always "begin with the end in mind," and so know exactly what you are going to deliver. If there is a detailed requirements document, mention this, too. This question helps determine the scope of the application. You can return to the answer of this question during your development to make sure you are staying the course and sticking to the agreed upon scope.

1. *"As is" or "to be"?* You need to know if there is a requirement to understand and model the current business environment (that is, the "as is" view), or to understand and model a proposed business environment (that is, the "to be" view). Sometimes the answer might be both, which means you first need to understand how things currently work before building something new. This question helps ensure you are capturing the correct time perspective.
2. *Is analytics a requirement?* Analytics, in informal terms, is the field of playing with numbers. That is, taking measurements such as **Gross Sales Amount** or **Inventory Count** and viewing them at different levels of granularity such as by day or year. If there is a requirement for analytics, at least part of your solution needs to be dimensional. Relational modeling focuses on business rules, and dimensional modeling focuses on business questions. This question will determine whether a relational or dimensional solution is needed.
3. *Who is the audience?* That is, who is the person or group who is the validator and can confirm your understanding of the CDM, and who will be the users of the CDM? It is a good policy with anything you produce to determine early on who will check your work (the validators) and who will be the recipient or users of your work. This question will ensure you choose the ideal display format for your conceptual data model. Note that if the validators and users vary considerably in their technical expertise, you may need more than one form for the CDM.

4. *Flexibility or simplicity?* In terms of design, there is always a balancing act between flexibility and simplicity. If you are leaning more towards flexibility, you will most likely use some generic terms such as **Event** instead of **Order** or **Person** instead of **Employee**. If your focus is more on simplicity, you will choose to use more of the business language such as **Order** and **Employee**.

Here is a sample answer for each of these questions for a university's student reporting system:

1. What is the application going to do?

This application is going to easily allow us to analyze key measurements concerning students. Today, much of our student reporting is very department specific and therefore the measurements and even meanings of key terms such as Student might vary across department, leading to inconsistency issues among these key measurements. This new application is designed to provide a broader, more enterprise view for student analysis.

2. "As is" or "to be"?

The application we are building is a brand-new system based upon several department-specific systems that are no longer meeting our needs. Therefore, we need a "to be" brand-new solution.

3. Is analytics a requirement?

Yes, we will need the ability to view key student metrics at different levels of detail such as by month and year.

4. Who is the audience?

Mary is the validator. She is the business analyst who will need to confirm our understanding of the conceptual data modeling phase. Project management will use our results going forward, as will all team members as an introduction into what the project is going to do.

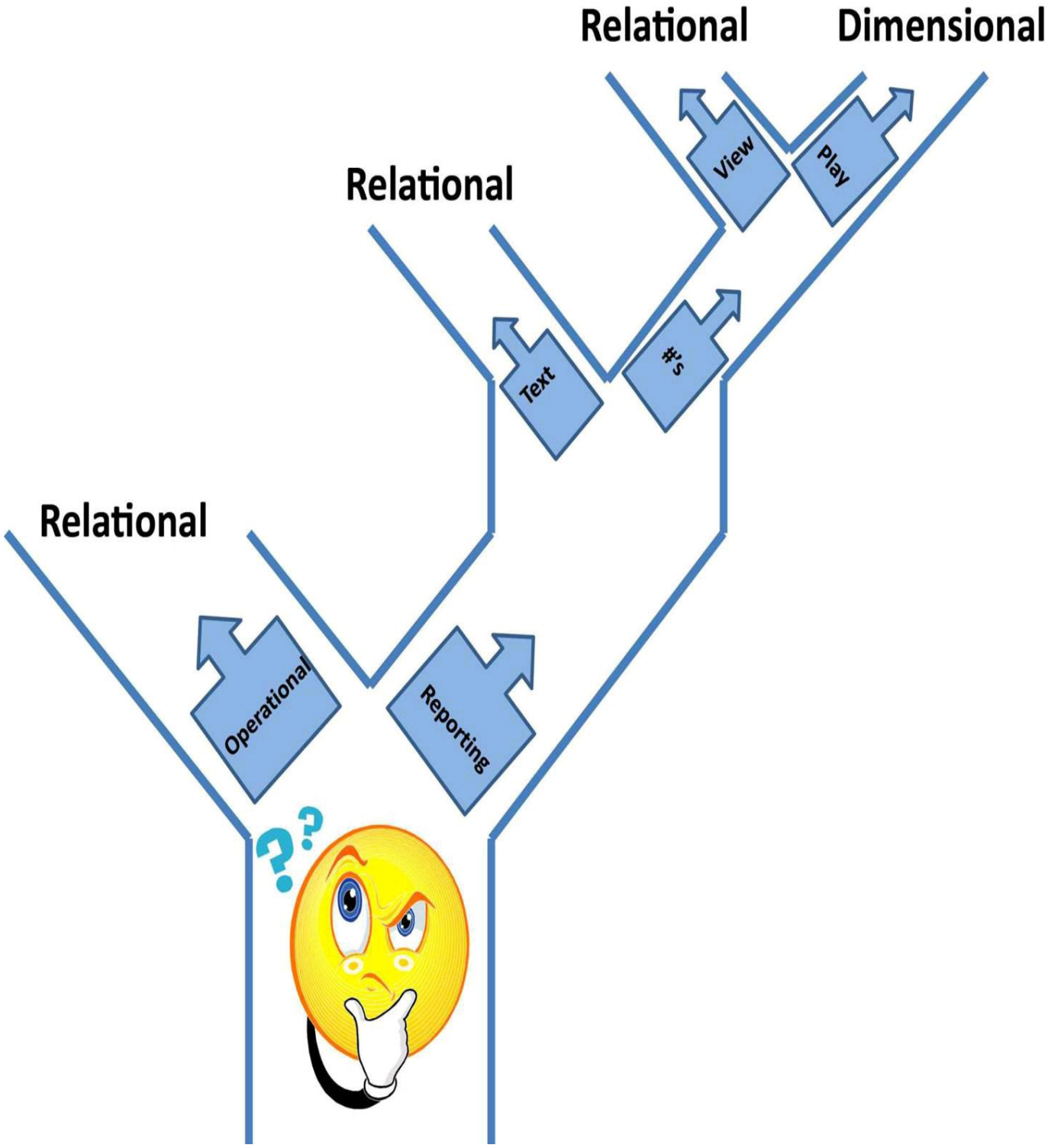
5. Flexibility or simplicity?

For the most part, simplicity should be given the priority. We want terms our users are familiar with. We need to speak their language and use their business terminology.

STEP 2: IDENTIFY AND DEFINE THE CONCEPTS

Now that we have direction, we can work with the business experts to identify the concepts within the scope of the application and come up with an agreed-upon definition for each concept.

Question number three, capturing whether analytics is a requirement, is a key question because it determines whether we need to take a relational or dimensional conceptual data modeling approach (or sometimes both a relational and dimensional approach). This is the thought process we go through in deciding whether a relational or dimensional data model is needed:



So the first decision is whether we are building an operational or reporting system. An operational system automates one or more business processes for an organization such as a claims processing system for an insurance company or an order entry system for a retail organization. A reporting system uses data from one or more operational systems to produce an evaluation of how part or all of the organization is performing. If we are building an operational system, go relational because relational captures business processes and an operational system automates business processes, so it is a perfect

fit. If building a reporting system, continue to the next fork in the road, which is whether there is a requirement to see text or numbers.

So, continuing down the path, if the only reporting requirements involve seeing text, such as names, codes, or descriptions, go relational. If there is a need to see numbers, continue to the next fork. The next fork in the road is whether these numbers are view only or going to be played with. If the requirement is just to view the numbers in a static fashion and not do any analysis with them, then go relational. If we are building a reporting system where there is a requirement to analyze and play with the numbers (such as to analyze **Gross Sales Amount** at a month level and then drill down to the date level), then go dimensional.

You can observe from this discussion that not all reporting systems are dimensional. In fact, a dimensional data model excels at analyzing measures (data elements that can be mathematically manipulated such as **Gross Sales Amount** or **Claims Count**). However, it can be less than optimal for all other types of reporting.

Relational data models are more flexible in general because they model business processes and dimensional models model business questions which can be more rigid and built to purpose. So, to summarize the decision process as to go relational or dimensional, the only time we are going to build dimensional data models is when we have a requirement to analyze numbers. For everything else, we go relational.

Another observation from this relational/dimensional decision process is that it has nothing to do with whether the resulting application will be built upon a relational database or a NoSQL database such as MongoDB. Once we decide relational or dimensional, then we build the conceptual, logical, and physical data models, and it is not until we reach the physical data model where we consider the database technology.

Concept terms and their definitions and relationships could be interpreted differently if we choose a relational verse dimensional path. In general, the definitions for terms on a relational data model are broader than those on a dimensional data model. On a dimensional data model, since the dimensions are often filtered for a particular usage and to answer a particular set of questions, the terms tend to be narrower in meaning and more specific than in relational data models. For example, if there is a dimensional data model whose purpose is to only analyze raw materials, the more general definition of product on the relational as “anything we buy or sell” becomes more specific on the dimensional by focusing only on raw materials as “anything we buy.”

Let’s now identify and define the concepts for both relational and dimensional.

For Relational

To identify and define the relational concepts, recall our definition of an entity as a noun or noun phrase that fits into one of six categories – who, what, when, where, why, or how. We can use these six categories to create a Concept Template for capturing the entities on our conceptual data model:

Concept Template

Who?	What?	When?	Where?	Why?	How?
1.	1.	1.	1.	1.	1.
2.	2.	2.	2.	2.	2.
3.	3.	3.	3.	3.	3.
4.	4.	4.	4.	4.	4.
5.	5.	5.	5.	5.	5.

To complete this template, list up to five concepts under each of these questions. Recall our earlier explanation of each entity category, which can help us complete each of the columns:

Category	Definition	Examples
Who	Person or organization of interest to the enterprise. That is, "*Who* is important to the business?" Often a *Who* is associated with a role such as Customer or Vendor.	Employee, Patient, Player, Suspect, Customer, Vendor, Student, Passenger, Competitor, Author
What	Product or service of interest to the enterprise. It often refers to what the organization makes that keeps it in business. That is, "*What* is important to the business?"	Product, Service, Raw Material, Finished Good, Course, Song, Photograph, Title
When	Calendar or time interval of interest to the enterprise. That is,	Time, Date, Month, Quarter, Year, Semester, Fiscal Period,

	"*When* is the business in operation?"	Minute
Where	Location of interest to the enterprise. Location can refer to actual places as well as electronic places. That is, "*Where* is business conducted?"	Mailing Address, Distribution Point, Website URL, IP Address
Why	Event or transaction of interest to the enterprise. These events keep the business afloat. That is, "*Why* is the business in business?"	Order, Return, Complaint, Withdrawal, Deposit, Compliment, Inquiry, Trade, Claim
How	Documentation of the event of interest to the enterprise. Documents record the events such as a Purchase Order recording an Order event. That is, "*How* does the business keep track of events?"	Invoice, Contract, Agreement, Purchase Order, Speeding Ticket, Packing Slip, Trade Confirmation

For each concept you identify in the Concept Template, write a definition that is at least a few sentences in length. Include examples and feel free to clarify concepts that are often frequently confused (e.g., you may want to explain the difference between **Customer** and **Consumer** in the **Customer's** definition).

For example, there is a bank that is replacing one of their core applications: the operational system that stores the account balances for each of its customers. Because it is an operational system, you will be building a relational conceptual data model. After meeting with the business users, you were able to complete the template on the facing page. Note that not all of the concepts identified in this spreadsheet will eventually appear on the conceptual data model. Sometimes very granular concepts such as **Account Open Date** make their first appearance on the logical data model instead of the conceptual.

Account Project Concepts

Who?	What?	When?	Where?	Why?	How?
1. Customer	1. Account	1. Account Open Date	1. Branch	1. Check Debit	1. Check
2.	2.	2.	2.	2. Deposit Credit	2. Deposit Slip
3.	3.	3.	3.	3. Interest Credit	3. Withdrawal Slip

4.	4.	4.	4.	4. Monthly Statement Fee	4. Bank Account Statement
5.	5.	5.	5.	5. Withdrawal Debit	5. Account Balance

Following is a full list of the concept definitions that you arrived at from your meetings with the business users. These definitions are provided to give you an idea of the level of detail needed to provide value for the people using your model (that is, a definition of **Account** which only says "The Customer Account" offers little if any value to the user):

Account	An account is an arrangement by which our bank holds funds on behalf of a customer. This arrangement includes the amount of funds we hold on behalf of a customer as well as a historical perspective of all of the transactions that have impacted this amount such as deposits and withdrawals. An account is structured for a particular purpose such as for stock investing, which is called a "brokerage account," for interest-bearing, which is called a "savings account," and for check writing, which is called a "checking account." An account can only be one of these types. That is, an account cannot be both checking and savings.
Account Balance	An account balance is a financial record of how much money a customer has with our bank at the end of a given time period such as someone's checking account balance at the end of a month. The account balance is impacted by many types of transactions including deposits and withdrawals. The account balance amount is restricted to just a single account. That is, if we wanted to know Bob the customer's net worth to our bank, we would need to sum the account balances for all of Bob's accounts.
Account Open Date	The day, month, and year that a customer first opens their account. This is the date that appears on the new account application form and is often not the same date as when the account first becomes active and useable. It may take 24-48 hours after the application is submitted for the account to be useable. The account open date can be any day of the week including a date when the bank is closed (such as if the customer submits the application using our website off hours).
Bank Account Statement	A periodic record of the events that have impacted the account. Events include withdrawals, deposits, etc. The bank account statement is usually issued monthly and includes the beginning account balance, a record of all events, and the ending account balance. Also listed are bank fees and if applicable, any interest accrued.

Branch	A division of our bank that is open to the public. A branch is a staffed facility, whether staffed virtually or staffed onsite. A branch does not include ATM-only facilities. A branch is service oriented, allowing our customers to perform many actions ranging from account services to home mortgage to brokerage.
Check	A check is a piece of paper that is equivalent to the stated amount in cash. It is a promise to pay someone the amount written on the check and to have this amount deducted from the check owner's (that is, the customer's) bank account. The concept of a check represents a filled in check as opposed to the concept of a check template. That is, if Bob writes a check for $5, the check captures this $5 amount and not the general concept that Bob has checks.
Check Debit	This is the result of cashing a check. The amount from the check is deducted from the customer's checking account. This is the event that adjusts the account for the check that was cashed.
Customer	A customer is a person or organization who may have one or more active accounts with our bank. Note that prospects (who do not yet have an account) are also considered customers. Note too that once the customer closes all of their accounts with us, they are no longer considered a customer.
Deposit Credit	This is the event of adding funds to an account. An example would be if someone brought $5 in to their branch and requested that it be added to their savings account.
Deposit Slip	This is the document that records how much should be added into an account. The deposit slip contains the account information along with how much should be added to the account.
Interest Credit	This is the amount of money the bank pays the customer for having the customer keep their money in the bank. Not all accounts pay interest, and the interest can vary based on many factors including the type of account.
Monthly Statement Fee	The amount of money charged each month to the customer to maintain an account.
Withdrawal Debit	An event where money is removed from the account. An example would be if someone took $5 out of their savings account.

Withdrawal Slip	The document recording money being removed from an account. The withdrawal slip contains the account information along with how much should be removed from the account.

For Dimensional

For dimensional, we need to determine the specific business questions that must be answered. For example, imagine that we work with the business analysts for a university and identify the following four questions.

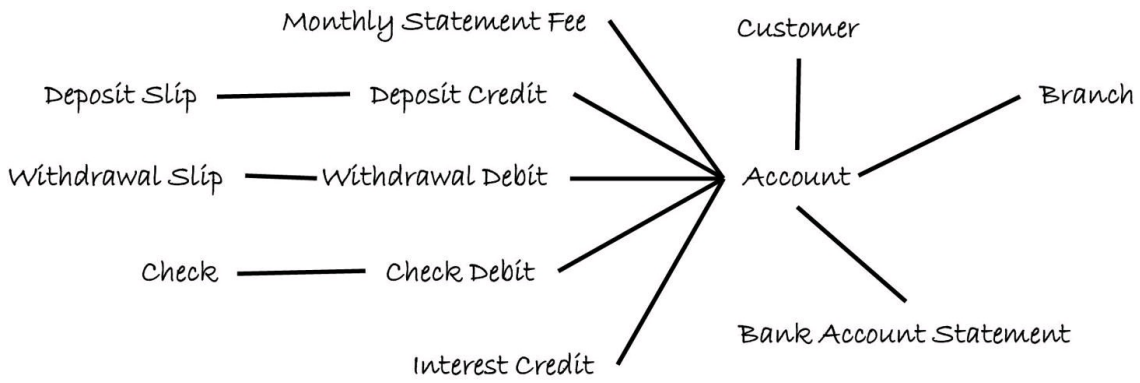
1. Show me the number of students receiving financial aid by department and semester for the last five years. [From Financial Aid Office]
2. Show me the number of students on full or partial scholarship by department and semester for the last four years. [From Accounting Department]
3. How many students graduated by department and semester over the last three years? [From Alumni Affairs]
4. How many students applied to the university over the last ten years? I want to compare applications from high school students vs. other universities. [From Admissions Department]

STEP 3: CAPTURE THE RELATIONSHIPS

Now we need to identify how these concepts relate to each other. Again, relational and dimensional each have different approaches.

For Relational

Relational is all about capturing the business rules, so our objective at the relational conceptual level is to determine which entities relate to each other and then articulate the rules. Let's assume we met with the business users and sketched the following on a whiteboard with them:



These concepts are the ones we identified from the prior step, with the exceptions of **Account Open Date** and **Account Balance**. We decided not to include these two concepts because they seemed very granular and therefore properties that would appear on the logical data model and not the conceptual.

Each line on this sketch represents a business rule in the form of a relationship between two concepts, so **Customer** and **Account** have a relationship, **Check** and **Check Debit** have a relationship, etc.

Once we've identified the fact that two entities have a relationship, we need to articulate what that relationship looks like and make it precise. To make it precise, we need to determine whether each entity in a relationship has a one or many participation with its related entity as well as whether each entity is required or optional in the relationship. We need to know participation and optionality for both entities in a relationship, leading to four questions to ask for each relationship.

If there is a relationship between **Entity A** and **Entity B**, here are the two participation questions I would ask:

- Can an **Entity A** be related to more than one **Entity B**?
- Can an **Entity B** be related to more than one **Entity A**?

And here are the two optionality questions I would ask:

- Can an **Entity A** exist without an **Entity B**?
- Can an **Entity B** exist without an **Entity A**?

In addition, we can also identify where we might add subtypes. Recall from Chapter 3 that subtyping groups the common properties of entities, while retaining what is special within each entity. Subtyping is an excellent way of communicating an entity lifecycle or for showing examples. For each entity, ask these two questions to

determine whether subtypes should be added to the model:

- Are there examples of this entity that would be valuable to show on the model, either for communication purposes or to enforce certain rules?
- Does this entity go through a lifecycle?

So for each relationship line on our model, we find ourselves asking up to eight questions: two on participation, two on optionality, and up to four questions on subtyping. For the model we just sketched, we can create a question template to make it easy to select “Yes” or “No” for the answer to each of these questions:

Can an Entity A be related to more than one Entity B?

Can an Entity B be related to more than one Entity A?

Can an Entity A exist without an Entity B?

Can an Entity B exist without an Entity A?

Are there examples of Entity A that would be valuable to show?

Are there examples of Entity B that would be valuable to show?

Does an Entity A go through a lifecycle?

Does an Entity B go through a lifecycle?

So after creating a template for our Account example, I asked a business expert each question and checked off the appropriate answer:

Can a Customer own more than one Account?	✓	
Can an Account be owned by more than one Customer?	✓	
Can a Customer exist without an Account?	✓	
Can an Account exist without a Customer?		✓
Are there examples of Customer that would be valuable to show?		✓
Are there examples of Account that would be valuable to show?	✓	
Does a Customer go through a lifecycle?	✓	
Does an Account go through a lifecycle?		✓
Can a Branch contain more than one Account?	✓	
Can an Account belong to more than one Branch?		✓

Can a Branch exist without an Account?	✓	
Can an Account exist without a Branch?		✓
Are there examples of Branch that would be valuable to show?		✓
Does a Branch go through a lifecycle?		✓
Can an Account generate more than one Bank Account Statement?	✓	
Can a Bank Account Statement be generated by more than one Account?	✓	
Can an Account exist without a Bank Account Statement?	✓	
Can a Bank Account Statement exist without an Account?		✓
Are there examples of Bank Account Statement that would be valuable to show?		✓
Does a Bank Account Statement go through a lifecycle?		✓
Can an Account incur more than one Monthly Statement Fee?	✓	
Can a Monthly Statement Fee be charged to more than one Account?		✓

Can an Account exist without a Monthly Statement Fee?	✓	
Can a Monthly Statement Fee exist without an Account?		✓
Are there examples of Monthly Statement Fee that would be valuable to show?		✓
Does a Monthly Statement Fee go through a lifecycle?		✓
Can an Account incur more than one Deposit Credit?	✓	
Can a Deposit Credit be credited to more than one Account?		✓
Can an Account exist without a Deposit Credit?	✓	
Can a Deposit Credit exist without an Account?		✓
Are there examples of Deposit Credit that would be valuable to show?		✓
Does a Deposit Credit go through a lifecycle?		✓
Can an Account incur more than one Withdrawal Debit?	✓	
Can a Withdrawal Debit be charged to more than one Account?		✓

Can an Account exist without a Withdrawal Debit?	✓	
Can a Withdrawal Debit exist without an Account?		✓
Are there examples of Withdrawal Debit that would be valuable to show?		✓
Does a Withdrawal Debit go through a lifecycle?		✓
Can an Account incur more than one Check Debit?	✓	
Can a Check Debit be charged to more than one Account?		✓
Can an Account exist without a Check Debit?	✓	
Can a Check Debit exist without an Account?		✓
Are there examples of Check Debit that would be valuable to show?		✓
Does a Check Debit go through a lifecycle?		✓
Can an Account incur more than one Interest Credit?	✓	
Can an Interest Credit be credited to more than one Account?		✓
Can an Account exist without an Interest Credit?	✓	

Can an Interest Credit exist without an Account?		✓
Are there examples of Interest Credit that would be valuable to show?		✓
Does an Interest Credit go through a lifecycle?		✓
Can a Deposit Slip record more than one Deposit Credit?	✓	
Can a Deposit Credit be recorded by more than one Deposit Slip?		✓
Can a Deposit Slip exist without a Deposit Credit?	✓	
Can a Deposit Credit exist without a Deposit Slip?		✓
Are there examples of a Deposit Slip that would be valuable to show?		✓
Does a Deposit Slip go through a lifecycle?		✓
Can a Withdrawal Slip record more than one Withdrawal Debit?	✓	
Can a Withdrawal Debit be recorded by more than one Withdrawal Slip?		✓
Can a Withdrawal Slip exist without a Withdrawal Debit?	✓	

Can a Withdrawal Debit exist without a Withdrawal Slip?		✓
Are there examples of Withdrawal Slip that would be valuable to show?		✓
Does a Withdrawal Slip go through a lifecycle?		✓
Can a Check lead to more than one Check Debit?		✓
Can a Check Debit be the result of more than one Check?		✓
Can a Check exist without a Check Debit?	✓	
Can a Check Debit exist without a Check?		✓
Are there examples of Check that would be valuable to show?		✓
Does a Check go through a lifecycle?		✓

I shaded in sets of rows to make it easier to read the questions for each relationship set. With the answers to these questions, we have enough information to build the conceptual data model. For example, we can model the answers to the first four questions between **Customer** and **Account**:

- Each **Customer** may own one or many **Accounts**.
- Each **Account** must be owned by one or many **Customers**.

Walk through the first four questions and see how it leads to this model. If you need a

refresher on cardinality (those symbols on the relationship lines), please refer back to Chapter 3.

Note that often after answering these questions, we uncover new entities. For example, because "Yes" was checked for *Are there examples of Account that would be valuable to show?*, we will most likely show several examples of **Account** as subtypes of **Account** on the model. Because "Yes" was checked for *Does a Customer go through a lifecycle?*, we will most likely show the lifecycle of **Customer** as subtypes of **Customer** on the model. Adding new subtypes means defining each subtype and also refining some of the questions that are asked to the business expert, which leads to refining some of the relationships on the resulting model. For example, if we add a **Checking Account** subtype to **Account**, it is possible that **Check Debit** just applies to **Checking Account** and not to all accounts such as **Savings Account**.

For Dimensional

For dimensional, we need to take the business questions we identified in the prior step and then create a grain matrix. A grain matrix is a spreadsheet where the measures from the business questions become columns and the dimensional levels from each business question become rows.

The purpose of a grain matrix is to efficiently scope analytic applications. It is possible to elicit hundreds of business questions, and after plotting these questions on a grain matrix, we make the observation that questions from different departments can actually be very similar to each other. By consolidating questions, we can scope applications that address the needs for more than one department.

So each business question is parsed in terms of the measure or measures it contains, and the levels of detail required for each measure. For example, recall the four university questions we identified in Step 2:

1. Show me the number of students receiving financial aid by department and semester for the last five years. [From Financial Aid Office]
2. Show me the number of students on full or partial scholarship by department and semester for the last four years. [From Accounting Department]
3. How many students graduated by department and semester over the last three years? [From Alumni Affairs]
4. How many students applied to the university over the last ten years? I want to compare applications from high school students vs other universities. [From Admissions Department]

Each of these questions comes from a different department yet focuses on the same measure, called **Student Count**. We would have to confirm that each area defines student the same way; if yes, we can complete a grain matrix such as the following:

	Student Count
Financial Aid Indicator	**1**
Semester	**1, 2, 3**
Year	**1, 2, 3, 4**
Department	**1, 2, 3**
Scholarship Indicator	**2**
Graduation Indicator	**3**
High School Application Indicator	**4**
University Application Indicator	**4**

So for question #1, we need to know the number of students receiving financial aid by department and semester for the last five years. We know the measure is **Student Count**. It is always a good idea to end each measure in a classword, which is the last part of the data element name. So for measures, this includes terms such as count, amount, quantity, volume, weight, percent, value, etc. I added **Financial Aid Indicator**

as a flag in this grain matrix so we can determine how many students are receiving financial aid. A value of "Yes" would indicate that the student is receiving financial aid, and a value of "No" would indicate that the student is not receiving financial aid. Put the question number in the cell that pertains to it so you can always trace back to the question; for example, question #1 requires the **Financial Aid Indicator** level. Then we also need to see student count by department and semester and year, so I put a "1" in each of these cells. Notice that I ignore value constraints such as "last five years" when filling in the grain matrix and only store at a year level. Ignoring value constraints makes it easier to see the connections across business questions since a question that needs something for the last three years and another question that needs that same something for the last five years will be the same question if we ignore the "three" or "five."

For question #2, I added the **Scholarship Indicator** as well as **Department**, **Semester**, and **Year**. For question #3, I added **Graduation Indicator**, and also the same other three levels of **Department**, **Semester**, and **Year**. For question #4, we need to distinguish high school students from students that applied from other universities, and therefore I added the two indicators, **High School Application Indicator** and **University Application Indicator**. I also added a reference to year because this requirement needs to see the last ten years.

STEP 4: DETERMINE THE MOST USEFUL FORM

Someone will need to review your work and use your findings during development, so deciding the most useful form is an important step. We know the users for the model after getting an answer to Strategic Question #4 from Step 1: *Who is our audience?* Who is the person or group who is the validator and can confirm our understanding of the CDM, and who will be the users of the CDM?

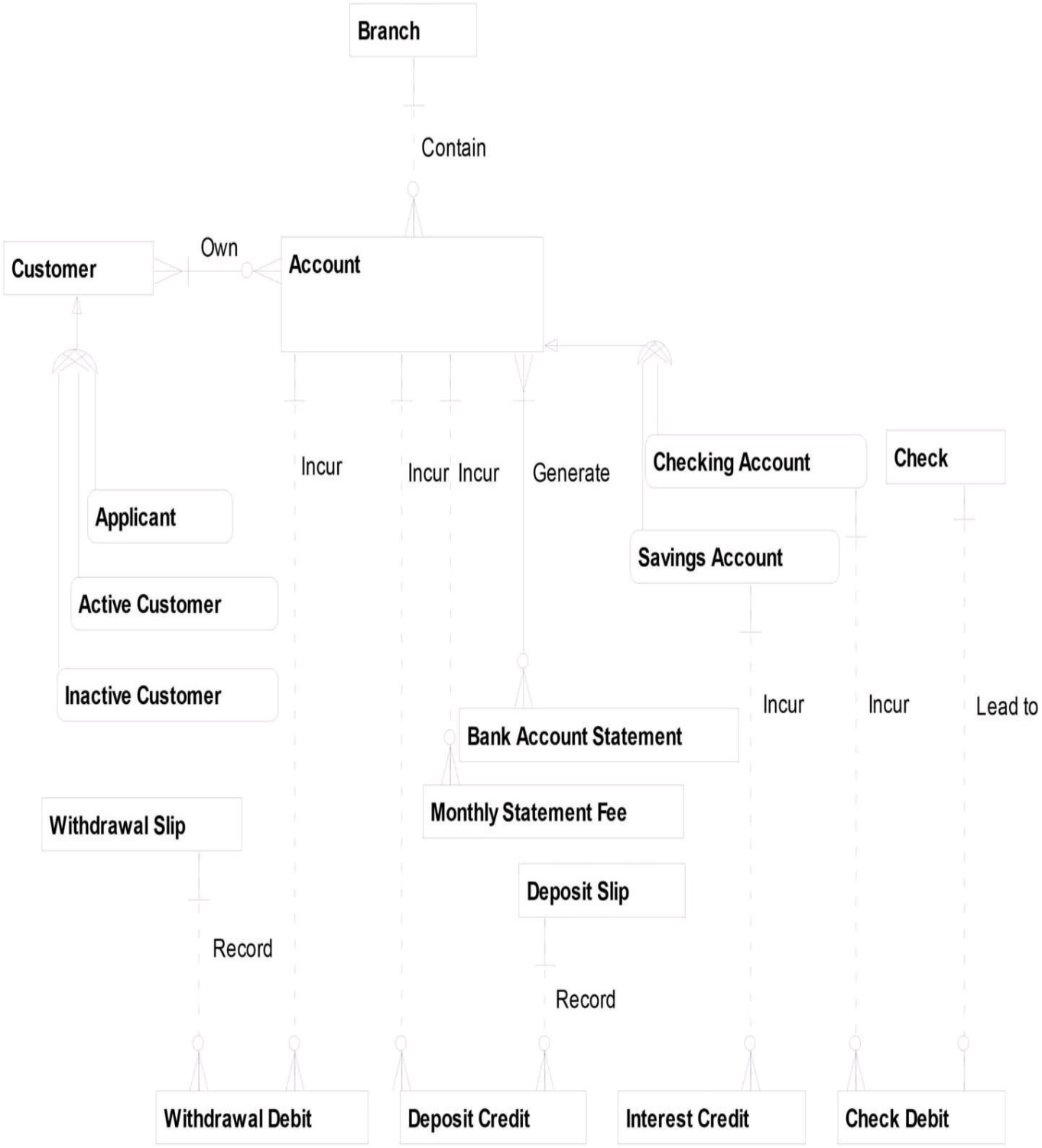
If the validator and users are already familiar with data modeling notation, the decision is an easy one: use the traditional data modeling notation they are comfortable with. However, very frequently at the conceptual level, the validator especially (and sometimes the users, too) are not familiar with traditional data modeling notation or simply don't want to see a data model. In these situations, be creative with how you display the model, coming up with a visual that the audience for the model would understand. I'll share with you some forms you can choose from for both relational and dimensional. However, the resulting form you come up with can vary from the examples that follow—and that's great, as long as you are thinking of your audience.

For Relational

There are many different forms we can choose from for the relational conceptual data

modeling deliverable. Let's look at three options.

Traditional Conceptual Data Model



Use traditional data modeling syntax if the validator and users are already familiar with data modeling notation. In our Account example, after modeling the answers to all of the questions from the prior step, we would have the CDM on the previous page.

Spend a few minutes going through the answers to the questions from the prior step and see how they translate into this model.

Note how some of the questions we ask turn into follow-up questions with refinement to the model such as how a "Yes" for *Are there examples of Account that would be valuable to show?* led us to digging deeper and uncovering the **Checking Account** and **Savings Account** subtypes. A "Yes" for *Does a Customer go through a lifecycle?* led us to identifying **Applicant**, **Active Customer**, and **Inactive Customer** as subtypes of **Customer**.

Business Assertions

If you feel the validators and users for this conceptual data model would prefer not to see a diagram, we can present the model as a series of business assertions:

- Each **Customer** may own one or many **Accounts**.
- Each **Account** must be owned by one or many **Customers**.

- Each **Customer** may be an **Applicant**, **Active Customer**, or **Inactive Customer**.
- Each **Applicant** is a **Customer**.
- Each **Active Customer** is a **Customer**.
- Each **Inactive Customer** is a **Customer**.

- Each **Account** may be a **Checking Account** or **Savings Account**.
- **Checking Account** is an **Account**.
- **Savings Account** is an **Account**.

- Each **Branch** may contain one or many **Accounts**.
- Each **Account** must belong to one **Branch**.

- Each **Account** may incur one or many **Withdrawal Debits**.
- Each **Withdrawal Debit** must be charged to one **Account**.

- Each **Account** may incur one or many **Deposit Credits**.
- Each **Deposit Credit** must be credited to one **Account**.

- Each **Account** may incur one or many **Monthly Statement Fees**.
- Each **Monthly Statement Fee** must be charged to one **Account**.

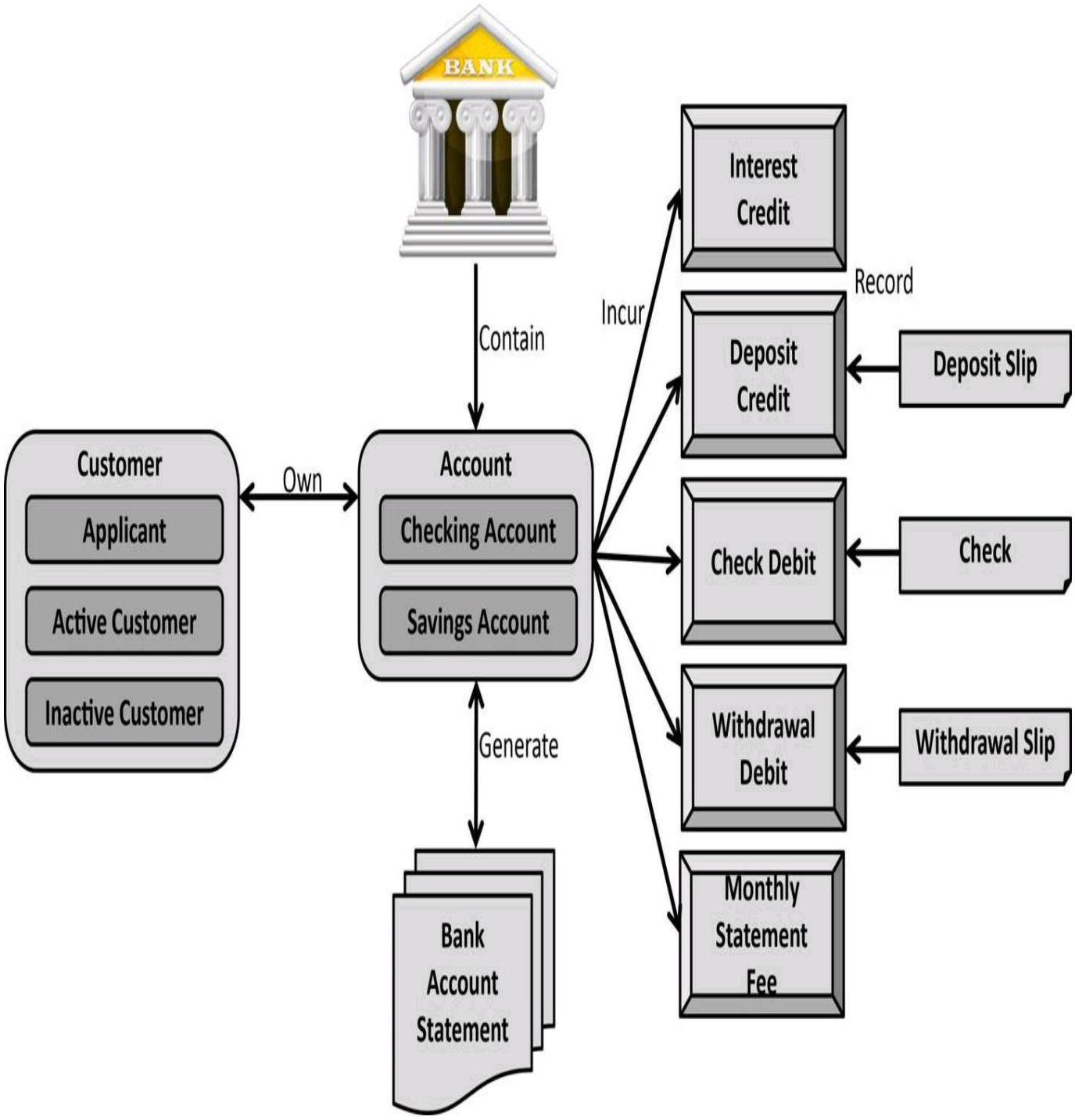
- Each **Account** may generate one or many **Bank Account Statements**.
- Each **Bank Account Statement** must be generated by one or many **Accounts**.

- Each **Checking Account** may incur one or many **Check Debits**.

- Each **Check Debit** must be charged to one **Checking Account**.
- Each **Check** may lead to one **Check Debit**.
- Each **Check Debit** must be the result of one **Check**.
- Each **Savings Account** may incur one or many **Interest Credits**.
- Each **Interest Credit** must be credited to one **Savings Account**.
- Each **Deposit Slip** may record one or many **Deposit Credits**.
- Each **Deposit Credit** must be recorded by one **Deposit Slip**.
- Each **Withdrawal Slip** may record one or many **Withdrawal Debits**.
- Each **Withdrawal Debit** must be recorded by one **Withdrawal Slip**.

Business Sketch

We can get very creative on how we format the data model, such as the business sketch of our account example on the following page where we use icons, sets, and pictures. There is an icon for large documents, such as for **Bank Account Statement**, an icon for small documents, such as for **Deposit Slip**, **Check**, and **Withdrawal Slip**, and an icon for transactions such as for the various credits, debits, and fee. The subtyping structures of **Customer** and **Account** are shown as sets, and there is a picture for the bank branch.



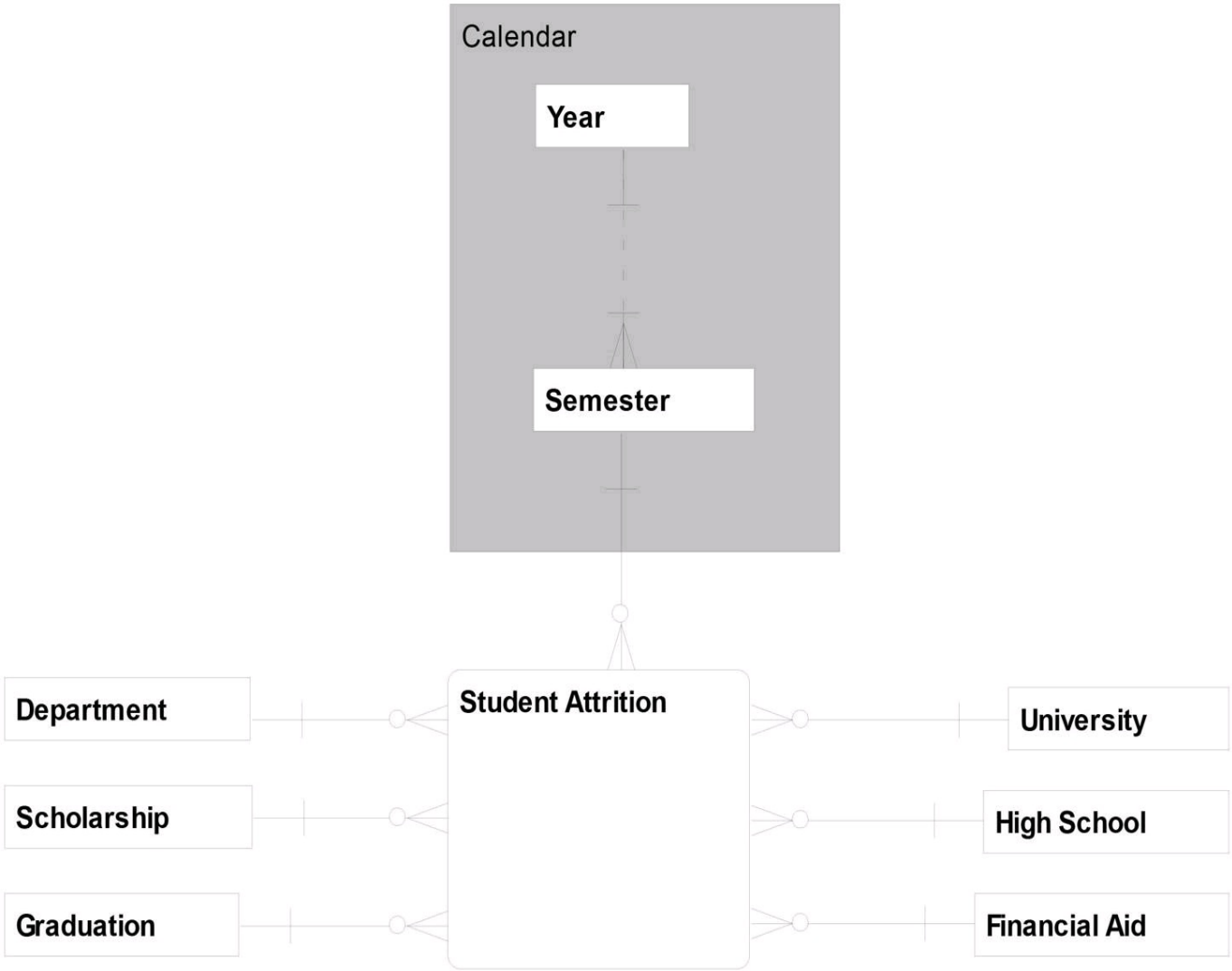
This model is less precise than the prior two because we are not showing cardinality but instead showing arrows from the parent entity to the child entity (or double-sided arrows for the many-to-many relationship between **Customer** and **Account** and between **Account** and **Bank Account Statement**). It is also less precise because we are connecting the arrows to **Account** instead of the **Account** subtypes (e.g., **Check Debit** connects to **Account** instead of **Checking Account**). However, this model could still be a great communication tool with the business, especially if there are solid definitions for each of these terms.

For Dimensional

Similar to relational data modeling, consider your audience (validators and users), and then choose the most appropriate form. Let's look at two options.

Traditional Conceptual Data Model

Use traditional data modeling syntax if the validator and users are already familiar with data modeling notation. In our University example, after modeling the grain matrix we would have this CDM:



Student Attrition is an example of a fact table (on a conceptual and logical data model often called a "meter"). A meter is an entity containing a related set of measures. These measures address one or more business processes such as **Sales**, **Claims**, or in this case **Student Attrition**. The meter is so important to the dimensional model that the name of the meter is often the name of the application–the **Sales** meter, the Sales Data Mart.

Department, **University**, and **Calendar** are examples of dimensions. A dimension is

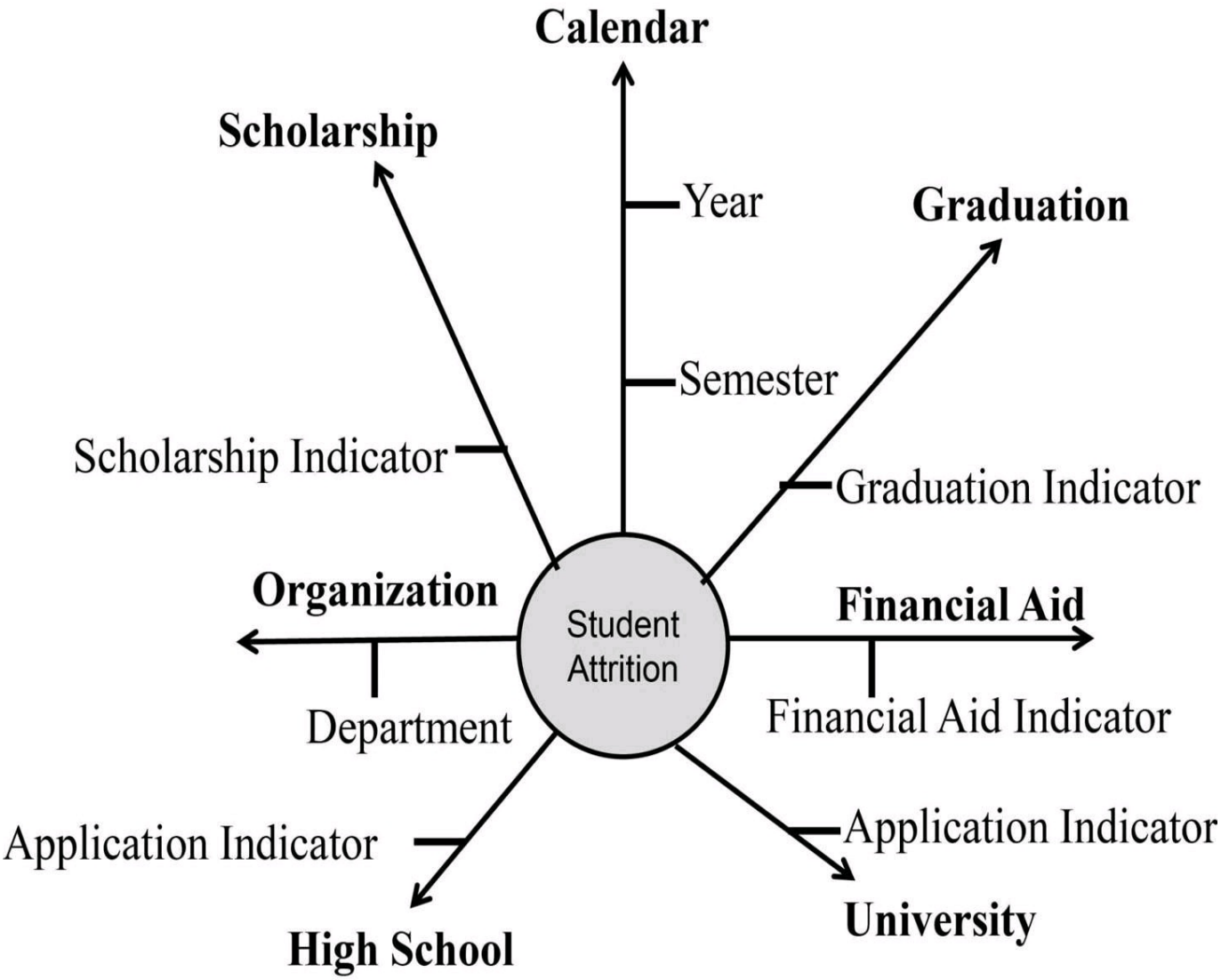
a subject whose purpose is to add meaning to the measures. All of the different ways of filtering, sorting, and summing measures make use of dimensions. A **Student Count** of *5* is meaningless unless it is associated with one or more dimensions.

A hierarchy is when a higher level can contain many lower levels, but a lower level can belong to, at most, one higher level. These higher levels indicate that we can view the measures in the meter at these levels as well. For example, we can view **Student Count** at either the **Year** or **Semester** levels. The **Calendar** dimension is an example of a two-level hierarchy.

If we felt that our audience would not understand the traditional modeling techniques, we could use a very business-friendly form called the Axis Technique.

Axis Technique

The Axis Technique is when you put the business process we are measuring in the center and each axis represents a dimension. The notches on each axis represent the levels of detail that are required to see the measures in the meter. This form works great when the audience has limited exposure to data modeling (or doesn't want to see the data model).



Student Attrition is the business process shown in the center of the circle and all of the different levels of detail we can view **Student Attrition** by are shown on the various axes.

STEP 5: REVIEW AND CONFIRM

Previously we identified the person or group responsible for validating the model. Now we need to show them the model and make sure it is correct. Often at this stage after reviewing the model we go back and make some changes and then show them the model again. This iterative cycle continues until the model is agreed upon by the validator and deemed correct so we can move on to the logical modeling phase.

EXERCISE 5: CONCEPTUAL DATA MODELING MINDSET

Recall the MongoDB document below we discussed in Chapter 3. Think of at least ten questions that you would ask during the conceptual data modeling phase based upon the data you see in this collection. See Appendix A for some of the questions I would

ask.

Order:

```
{  orderNumber : “4839-02”,
                    orderShortDescription : “Professor review copies of several titles”,
   orderScheduledDeliveryDate : ISODate(“2014-05-15”),
   orderActualDeliveryDate : ISODate(“2014-05-17”),
   orderWeight : 8.5,
   orderTotalAmount : 19.85,
   orderTypeCode : “02”,
   orderTypeDescription : “Universities Sales”,
   orderStatusCode : “D”,
   orderStatusDescription : “Delivered”,
   orderLine :
    [ {  productID : “9781935504375”,
      orderLineQuantity : 1
    },
                                                {     productID : “9781935504511”,
      orderLineQuantity : 3
    },
    {  productID : “9781935504535”,
      orderLineQuantity : 2
    } ] }
```

Key Points

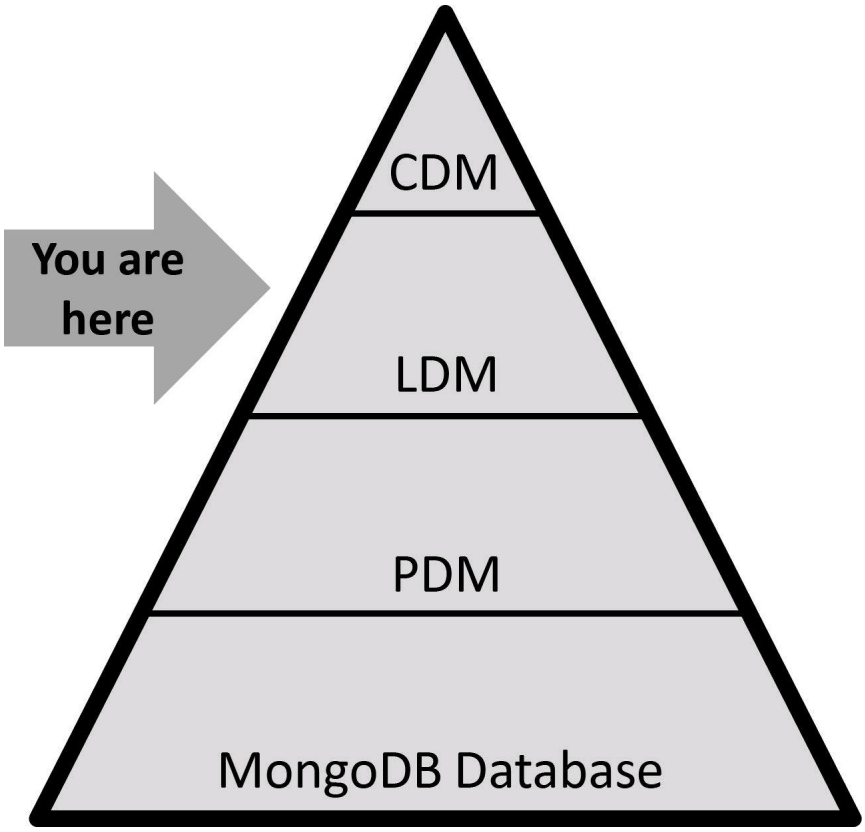
- Conceptual data modeling is the process of capturing the satellite view of the business requirements.
- The end result of the conceptual data modeling phase is a conceptual data model (CDM) that shows the key concepts needed for a particular application development effort.
- A concept is a key term that is both basic and critical to your audience.

- The main reason for conceptual data modeling is to get the "big picture" and understand the scope and needs of the project. Having a common level of understanding leads to refining scope and direction, raising key issues early in the process, and defining the most important terms.

- There are five steps to conceptual data modeling: ask the five strategic questions, identify and define the concepts, capture the relationships, determine the most useful form, and review and confirm.

- Be creative on the form you choose for the conceptual data model. Think of your users and validators.

Chapter 6
Logical Data Modeling



Logical data modeling is the process of capturing the detailed business solution. The logical data model looks the same regardless of whether we are implementing in MongoDB or Oracle.

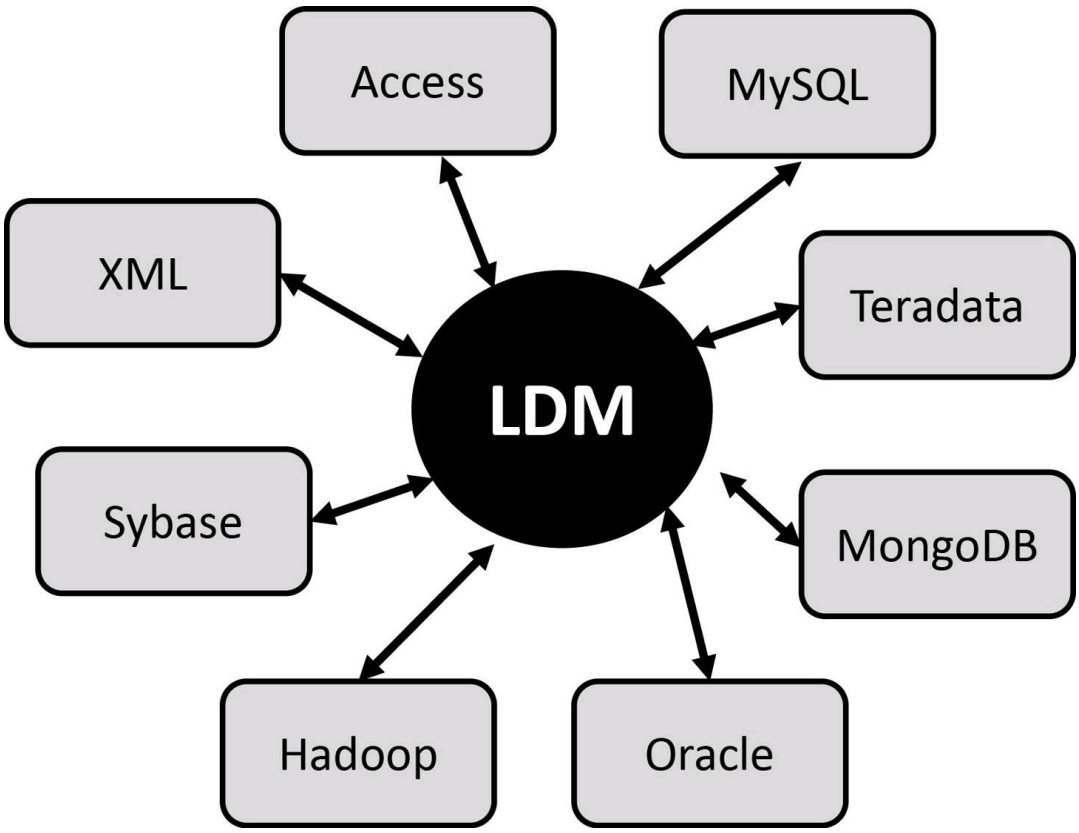
During conceptual data modeling we might learn, for example, what the terms, business rules, and scope would be for a new order entry system. Logical data modeling is taking these findings down to the next level and understanding all of the data requirements for the order entry system. For example, conceptual data modeling

will reveal that a **Customer** may place many **Orders**. Logical data modeling will uncover all of the details behind **Customer** and **Order**, such as the customer's name, their address, the order number, and what is being ordered. During logical data modeling, questions or issues may arise having to do with specific hardware or software such as:

- What should the collections look like?
- What is the optimal way to do sharding?
- How can we make this information secure?
- How should we store history?
- How can we answer this business question in less than 300 milliseconds?

These questions focus on technology concerns such as performance, data structure, and how to implement security and history. These are all physical considerations, and therefore are important to address during the physical data modeling stage, and not during logical data modeling. It is OK to document questions like these if they come up during logical data modeling, just make sure no time is spent addressing these questions until after the logical data modeling process is complete. Once the logical data modeling phase is complete, you will have more knowledge about what is needed, and will be able to make intelligent physical data modeling decisions including answering questions like these correctly.

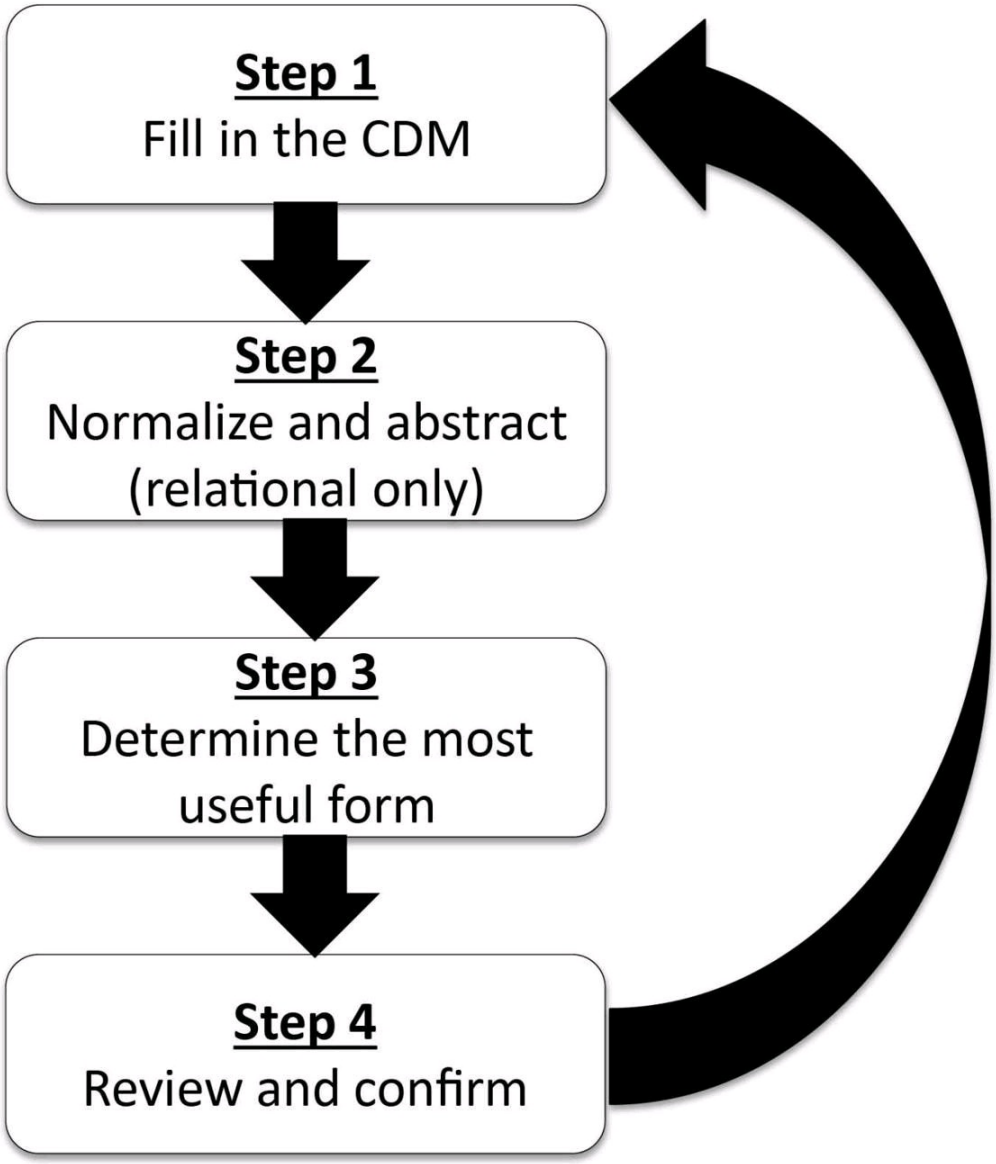
The logical data model is also a great mapping point when multiple technologies exist. The logical data model will show one technology-independent perspective, and then we can map this business perspective to each of the various technology perspectives, similar to a hub in a bicycle tire:



When new technologies come along, we can use the logical data model as a starting point and therefore save an incredible amount of time in building the application, as well as improving consistency within the organization.

Logical Data Modeling Approach

Logical data modeling involves completing the following four steps:



We can build upon the work done in conceptual data modeling by filling in the entities on the conceptual data model. This means identifying and defining the properties for each concept (Step 1). For relational only, we normalize and optionally abstract (Step 2). For dimensional, skip Step 2. Then similar to conceptual data modeling, we determine the most useful form (Step 3). As a final step, review your work to confirm we have the data requirements modeled correctly (Step 4). Frequently action items come out of the review and confirm step where we need go back to Step 1 and refine our concepts. Expect iteration through these steps, until the logical data model is confirmed and then we can start the physical. Let's talk more about each of these four

steps.

STEP 1: FILL IN THE CDM

We fill in the CDM by identifying and defining the properties of each concept. The two templates we complete at this step, the Properties Template and Property Characteristics Template, are useful for both relational and dimensional efforts.

The Properties Template captures the properties (also known as attributes or data elements) of each conceptual entity:

Properties Template

	Entity A	Entity B	Entity C
Name			
Text			
Amount			
Date			
Code			
Quantity			
Number			
Identifier			

Indicator

Rate

Percent

Complex

Each column heading represents one of the conceptual entities, and each row represents a grouping of properties by classword. Recall that a classword is the last part of the data element name, which represents a high level domain or category. Within each cell, you can list the properties for the conceptual entity that have that classword in common. For example, for the conceptual entity **Order** you can list the two properties **Order Gross Amount** and **Order Net Amount** next to the **Amount** cell, **Order Entry Date** and **Order Delivery Date** next to the **Date** cell, etc. Realize the initial names we use for each property may change as we learn more about each property.

To identify the properties for each conceptual entity, ask the question, "What *[Classword]* do I need to know about *[Conceptual Entity]*?" For example, "What **Dates** do I need to know about **Order**?"

Here are examples for each of the classwords, along with a brief description:

- Name. A textual value by which a thing, person, or concept is known. Examples: **Company Name**, **Customer Last Name**, **Product Last Name**
- Text. An unconstrained string of characters or any freeform comment or notes field. Examples: **Email Body Text**, **Tweet Text**, **Order Comments Text**
- Amount. A numeric measurement of monetary value in a particular currency such as dollars or Euros. Examples: **Order Total Amount**, **Employee Salary Amount**, **Produce Retail Price Amount**
- Date. A calendar date. Examples: **Order Entry Date**, **Consumer Birth Date**,

Course Start Date

- Code. A shortened form representing a descriptive piece of business information. Examples: **Company Code**, **Employee Gender Code**, **Currency Code**
- Quantity. A measure of something in units. Quantity is a broad category containing any property that can be mathematically manipulated with the exception of currencies, which are Amounts. Quantity includes counts, weights, volumes, etc. Examples: **Order Quantity**, **Elevator Maximum Gross Weight**, **Claim Count**
- Number. Number can be misleading as it is usually a business key (that is, a business user's way of identifying a concept). Number cannot be mathematically manipulated and often contains letters and special characters such as the hyphen in the case of a telephone number. Examples: **Social Security Number**, **Credit Card Number**, **Employee Access Number**
- Identifier. A mandatory, stable, and unique property of an entity that is used to identify instances of that entity. Examples: **Case Identifier**, **Transaction Identifier**, **Product Identifier**
- Indicator. When there are only two values such as *Yes* or *No*, *1* or *0*, *True* or *False, On* or *Off*. Sometimes called a "flag." Examples: **Student Graduation Indicator**, **Order Taxable Indicator**, **Current Record Indicator**
- Rate. A fraction indicating a proportion between two dissimilar things. Examples: **Employee Hourly Rate**, **State Tax Rate**, **Unemployment Rate**
- Percent. A ratio where 100 is understood as the denominator. Examples: **Ownership Percent**, **Gross Sales Change Percent**, **Net Return Percent**
- Complex. Anything that is not one of the above categories. This can include music, video, photographs, scanned images, documents, etc. Examples: **Employee Photo JPEG**, **Contract Signed PDF**, **Employee Phone Conversation MPP**

The second template, the Property Characteristics Template, captures important metadata on each property:

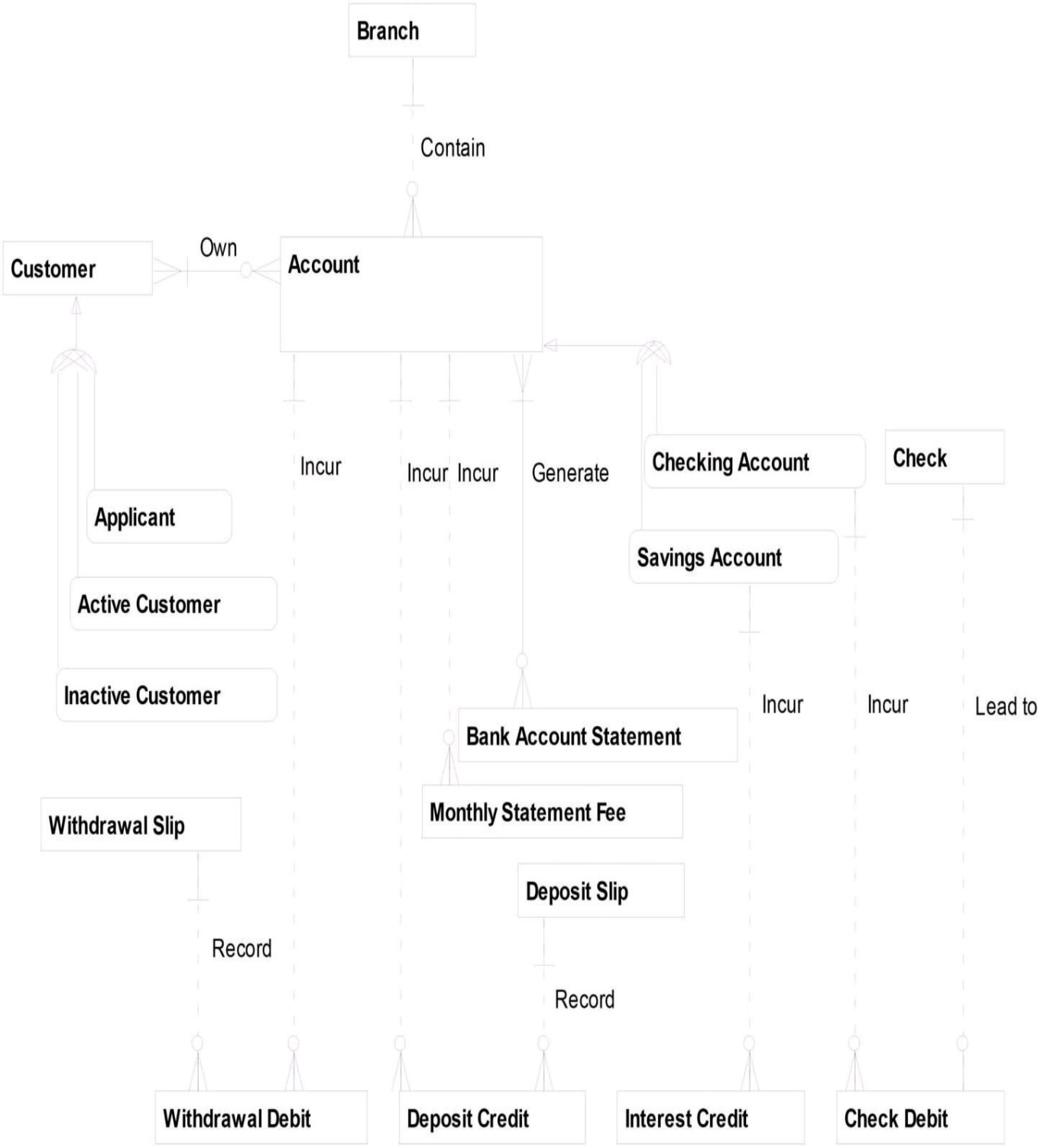
Property Characteristics Template

Property Definition Sample Values Format Length Source Transformations

To complete the Property Characteristics Template, enter the name for each property in the Property column (you can copy and paste from the Property Template). Then add the definition. I usually like to see a definition that is at least a few sentences long. I know the template doesn't leave much room for text, but in a landscape format you will have more room. If you think it would be beneficial to show some sample values, you can add those too. I often find with codes, such as **Country Code**, that it makes sense to add a few sample values. Format can be date, character, integer, etc., and the Length is how long the data element is. Optionally, you can include where the property comes from (the Source column) and any Transformations that might be required to bring data from the source into this property. Transformations are useful to capture for derived properties such as for **Gross Sales Amount**.

For Relational

Recall our relational CDM from Chapter 5:



After meeting with the business experts and studying existing documentation, we completed the following Properties Template for this CDM. (Note that to save column space, I grouped all of the credits and debits under **Transaction** and all of the documents, such as slips and statements, under **Document**. To save row space, I removed those classword rows where there were no properties identified. Note also that I am showing only a handful of the actual attributes that would come out of this

process.)

Account Project Properties

	Account	Customer	Branch	Transaction	Document
Name	Account Name	Customer Name, Favorite Sports Team			
Text			Branch Address Text		
Amount	Minimum Balance Amount			Monthly Fee Amount, Deposit Amount, Withdrawal Amount	Check Amount
Date	Account Open Date	Customer Birth Date, Applicant Date, Inactive Date		Interest Credit Amount, Interest Credit Date	Check Date, Deposit Date, Withdrawal Date
Code			Branch Code, Branch Size Code		
Quantity	Free Checks Per Month				
Number	Account Number	Social Security Number			Check Number, Deposit Number, Withdrawal Number

Rate	Credit Rating

And now we will fill in the Property Characteristics Template for each of these properties. To save space, I am only going to fill this in for five of the attributes from the Property Template above, but on an actual project you would complete this for all attributes:

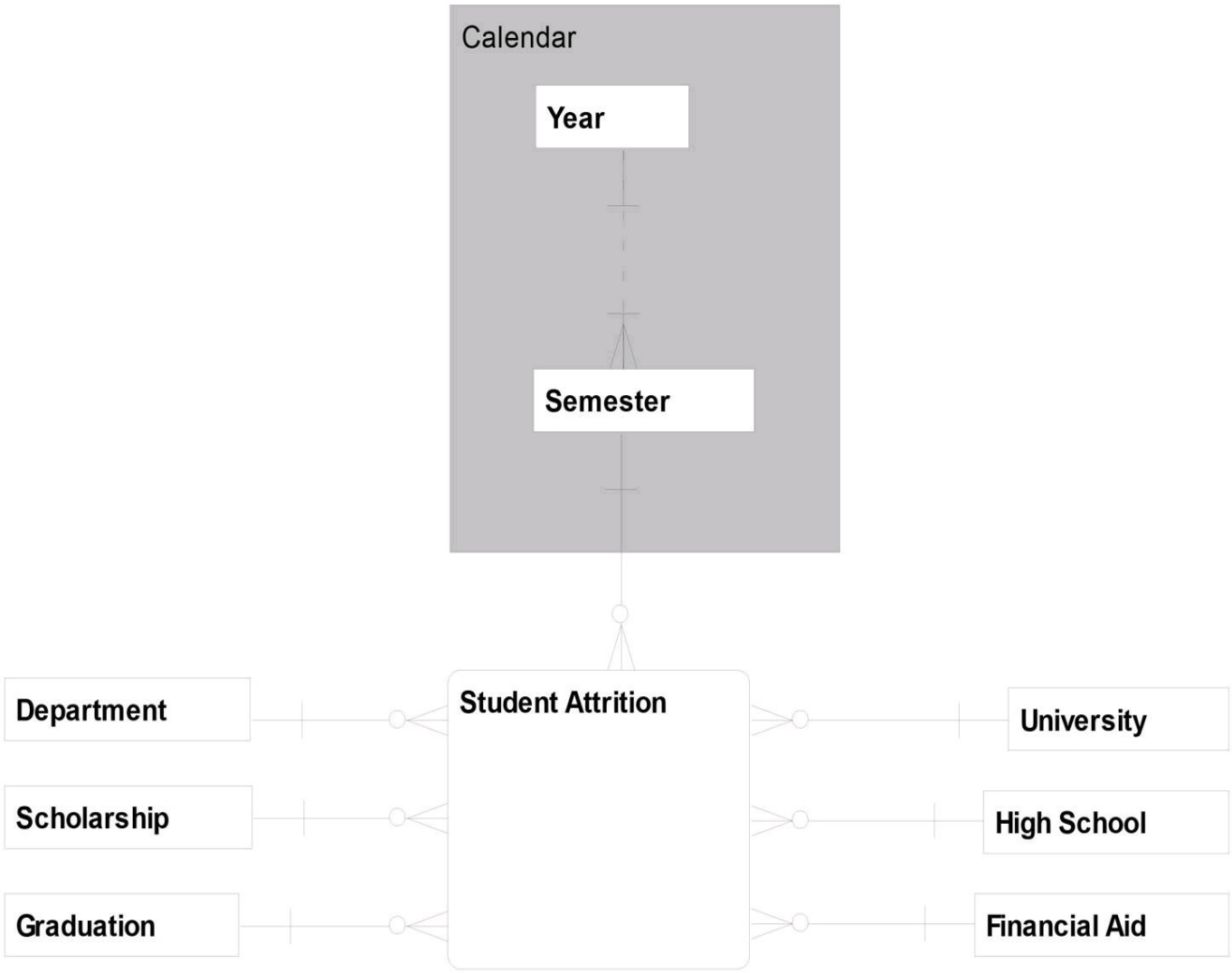
Account Project Property Characteristics

Property	Definition	Sample Values	Format	Length	Source	Transformations
Account Number	The unique number assigned to each account. This number is unique across branches. Our bank employees and our customers have knowledge of this number.	17382-112	Char	8	New AccountSystem	Not applicable
Favorite Sports Team	The favorite sports team of the customer, which is captured to allow the bank employees to have casual conversations with the customer.	Mets Giants Flyers	Char	100	CRM	Not applicable
Inactive Date	The date captured for the point where a customer has not had any transactions for a ten-year period.	03-12-2014	Date		Not applicable	Look at today's date and the customer's last transaction date. If the difference is greater than ten years, insert today's date.
Check Number	The unique number assigned to each check within a checking account. This is the number pre-printed on the upper right hand corner of the check.	123 134	Num	3	Acct Txn App	Not applicable

With-drawal Amount	The amount taken out of an account as initiated by completing a withdrawal slip.	$50.29	Dec	15,2	Acct Txn App	Not applicable

For Dimensional

The same two templates we just described for relational also apply to dimensional. Recall our dimensional CDM from the last chapter:



On the facing page is the completed Properties Template for this CDM. To save row space, I removed those classword rows where there were no properties identified. Note also that I am showing only a handful of the actual attributes that would come out of this process. I also combined all of the characteristics for viewing student attrition under the column Criteria instead of showing each indicator in a separate column.

Student Attrition Application Properties

	Department	Calendar	Criteria	Attrition
Name	Department Name			
Text		Year Full Name, Semester Name		
Code	Department Code	Year Code, Semester Code		
Number		Semester Sequence Number		Student Count
Indicator			Scholarship Indicator, Graduation Indicator, University Application Indicator, High School Application Indicator, Financial Aid Indicator	

And now we will fill in the Property Characteristics Template for each of these properties. To save space, I am only going to fill this in for the measure **Student Count** from the Property Template above, but on an actual project, you would complete this for all attributes:

Student Attrition Application Property Characteristics

Property	Definition	Sample Values	Format	Length	Source	Transformations
Student Count	The number of people currently enrolled in a program (both degree and non-degree).	50	Num	8	Student Reg App	Derived when loading data from the data warehouse.

STEP 2: NORMALIZE AND ABSTRACT (RELATIONAL ONLY)

Normalization and abstraction are techniques for properly assigning data elements to entities on the relational data model. Normalization is a mandatory technique, and abstraction is an optional technique.

Normalization

When I turned 12, I received a trunk full of baseball cards as a birthday present from my parents. I was delighted, not just because there might have been a Hank Aaron or Pete Rose buried somewhere in that trunk, but because I loved to organize the cards. I categorized each card according to year and team. Organizing the cards in this way gave me a deep understanding of the players and their teams. To this day, I can answer many baseball trivia questions.

Normalization, in general, is the process of applying a set of rules with the goal of organizing *something*. I was normalizing the baseball cards according to year and team. We can also apply a set of rules and normalize the attributes within our organizations. The rules are based on how the business works, and the relational data model captures the business rules, which is why normalization is the primary technique used in building the relational logical data model.

Just as those baseball cards initially lay unsorted in that trunk, our companies have huge numbers of attributes spread throughout departments and applications. The rules applied to normalizing the baseball cards entailed first sorting by year and then by team within a year. The rules for normalizing our attributes can be boiled down to a single sentence: Make sure every attribute is single-valued and provides a fact completely and only about its primary key. Single-valued means an attribute must contain only one piece of information. If **Consumer Name** contains **Consumer First Name** and **Consumer Last Name**, for example, we must split **Consumer Name** into two attributes – **Consumer First Name** and **Consumer Last Name**. Provides a fact means that a given primary key value will always return no more than one of every attribute that is in the same entity with this key. If a Customer Identifier value of *123*, for example, returns three customer last names (*Smith*, *Jones*, and *Roberts*), **Customer Last Name** violates this part of the normalization definition. Completely means that the minimal set of attributes that uniquely identify an instance of the entity is present in the primary key. Only means that each attribute must provide a fact about the primary key and nothing else. That is, there can be no hidden dependencies, so we would need to remove derived attributes.

We cannot determine if every attribute is single-valued and provides a fact completely and only about its primary key unless we understand the data. To understand the data,

we usually need to ask lots of questions. Therefore a second definition for normalization is: *A formal process of asking business questions*. Even for an apparently simple attribute such as Phone Number**,** for example, we can ask many questions:

- Whose phone number is this?
- Do you always have to have a phone number?
- Can you have more than one phone number?
- Do you ever recognize the area code as separate from the rest of the phone number?
- Do you ever need to see phone numbers outside a given country?
- What type of phone number is this? That is, is it a fax number, mobile number, etc.?
- Does the time of day matter? For example, do we need to distinguish between the phone number to use during working hours and outside working hours? Of course, that would lead to a discussion on what we mean by "working hours."

To ensure that every attribute is single-valued and provides a fact completely and only about its primary key, we apply a series of rules in small steps, where each step (or level of normalization) checks something that moves us towards our goal. Most data professionals would agree that the full set of normalization levels is the following:

- first normal form (1NF)
- second normal form (2NF)
- third normal form (3NF)
- Boyce/Codd normal form (BCNF)
- fourth normal form (4NF)
- fifth normal form (5NF)

1NF is the lowest level and 5NF the highest. Each level of normalization includes the lower levels of rules that precede it. If a model is in 5NF, it is also in 4NF, BCNF, and so on. Even though there are higher levels of normalization than 3NF, many

interpret the term *normalized* to mean 3NF. This is because the higher levels of normalization (that is, BCNF, 4NF, and 5NF) cover specific situations that occur much less frequently than the first three levels. Therefore, to keep things simple, let's focus on only first through third normal forms.

Initial Chaos

I would describe the trunk of baseball cards I received as being in a chaotic state as there was no order to the cards—just a bunch of cards thrown in a large box. I removed the chaos by organizing the cards. The term *chaos* can be applied to any unorganized pile including attributes. We may have a strong understanding of each of the attributes, such as their name and definition, but we don't know to which entity the attribute should be assigned. When I picked out a 1978 Pete Rose from the baseball card box and put this card in the 1978 pile, I started bringing order were there was chaos – similar to assigning **Customer Last Name** to the customer pile (called the **Customer** entity).

Let's walk through an example. Here's a bunch of what appears to be employee attributes:

Employee

Employee Identifier Department Code
Phone Number 1 Phone Number 2 Phone Number 3 Employee Name Department Name Employee Start Date Employee Vested Indicator

Often definitions are of poor quality or missing completely, so let's assume that this is the case with this **Employee** entity. We are told, however, that **Employee Vested Indicator** captures whether an **Employee** is eligible for retirement benefits – a value of *Y* for "yes" means the employee is eligible, and a value of *N* for "no" means the employee is not eligible. This indicator is derived from the employee's start date. For example, if an employee has worked for the company for at least five years, then this indicator contains the value *Y*, which means this employee is eligible for retirement benefits.

What is lacking at this point, and what will be solved though normalization, is assigning these attributes to the right entities.

It is very helpful to have some sample values for each of these attributes, so let's assume this spreadsheet is a representative set of employee values:

Emp Id	Dept Cd	Phone 1	Phone 2	Phone 3	Emp Name	Dept Name	Emp Start Date	Emp Vested Ind
123	A	973-555-1212	678-333-3333	343-222-1111	Henry Winkler	Data Admin	4/1/2012	N
789	A	732-555-3333	678-333-3333	343-222-1111	Steve Martin	Data Admin	3/5/2007	Y
565	B	333-444-1111	516-555-1212	343-222-1111	Mary Smith	Data Warehouse	2/25/2006	Y
744	A	232-222-2222	678-333-3333	343-222-1111	Bob Jones	Data Admin	5/5/2011	N

First Normal Form (1NF)

Recall that the series of rules can be summarized as: *Every attribute is single-valued and provides a fact completely and only about its primary key*. First Normal Form (1NF) is the "single-valued" part. It means that for a given primary key value, we can find, at most, one of every attribute that depends on that primary key.

Ensuring each attribute provides a fact about its primary key includes addressing repeating groups and multi-valued attributes. Specifically, the modeler needs to:

- **Move repeating attributes to a new entity**. When there are two or more of the same attribute in the same entity, they are called repeating attributes. The reason repeating attributes violate 1NF is that for a given primary key value, we are getting more than one value back for the same attribute. Repeating attributes often take a sequence number as part of their name, such as phone number in this employee example. We can find ourselves asking many questions just to

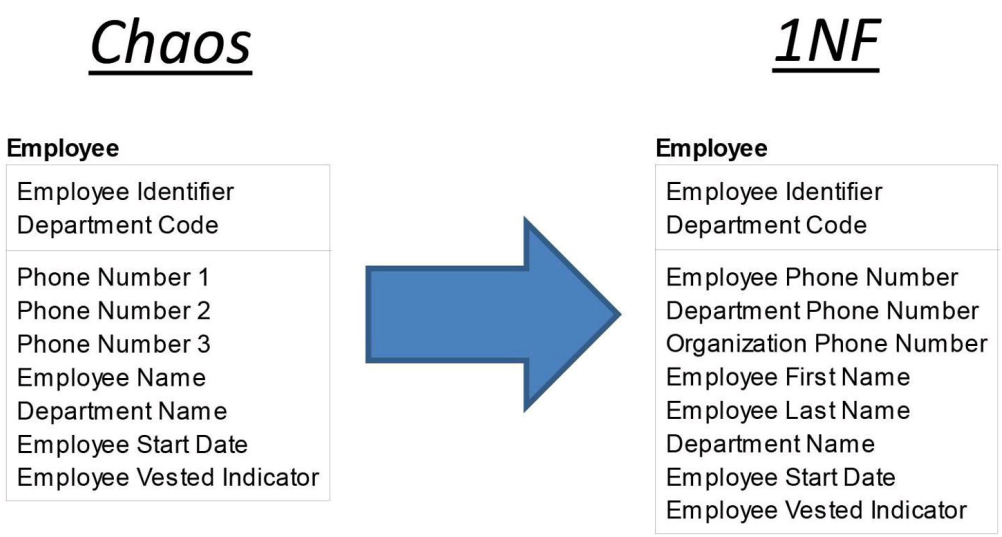
determine if there are any repeating attributes we need to address. We can have a question template such as "Can a [[insert entity name here]] have more than one [[insert attribute name here]]?" So these are all valid questions in our employee example:

- Can an **Employee** have more than one **Employee Identifier**?
- Can an **Employee** have more than one **Department Code**?
- Can an **Employee** have more than one **Phone Number**?
- Can an **Employee** have more than one **Employee Name**?
- Can an **Employee** have more than one **Department Name**?
- Can an **Employee** have more than one **Employee Start Date**?
- Can an **Employee** have more than one **Employee Vested Indicator**?

- **Separate multi-valued attributes**. *Multi-valued* means that within the same attribute we are storing at least two distinct values. There are at least two different business concepts hiding in one attribute. For example, **Employee Name** may contain both a first name and last name. **Employee First Name** and **Employee Last Name** can be considered distinct attributes, and therefore *Henry Winkler,* stored in Name, is multi-valued, because it contains both *Henry* and *Winkler*. We may find ourselves asking many questions just to determine if there are any multi-valued attributes we need to identify. We can have another question template, such as "Does a [[insert attribute name here]] contain more than one piece of business information?" So these are all valid questions in our employee example:

- Does an **Employee Identifier** contain more than one piece of business information?
- Does a **Department Code** contain more than one piece of business information?
- Does a **Phone Number** contain more than one piece of business information?
- Does an **Employee Name** contain more than one piece of business information?
- Does a **Department Name** contain more than one piece of business information?
- Does an **Employee Start Date** contain more than one piece of business information?
- Does an **Employee Vested Indicator** contain more than one piece of business information?

After answering these questions based on the earlier set of sample data, and by talking with a business expert, we can update the model:



We learned that although **Phone Number 1**, **Phone Number 2**, and **Phone Number 3** appear as repeating attributes, they are really three different pieces of information based upon the sample values we were given. **Phone Number 3** contained the same value for all four employees, and after validating with the business expert we learned that this is the organization's phone number. **Phone Number 2** varied by department, so this attribute was renamed to **Department Phone Number. Phone Number 1** is different for each employee, and we learned that this is the **Employee Phone Number**. We also were told that **Employee Name** does contain more than one piece of information, and therefore it should be split into **Employee First Name** and **Employee Last Name**.

Second Normal Form (2NF)

Recall that the series of rules can be summarized as: *Every attribute is single-valued and provides a fact completely and only about its primary key*. First Normal Form (1NF) is the "single-valued" part. Second Normal Form (2NF) is the "completely" part. This means each entity must have the minimal set of attributes that uniquely identifies each entity instance.

As with 1NF, we will find ourselves asking many questions to determine if we have the minimal primary key. We can have another question template such as: "Are all of the attributes in the primary key needed to retrieve a single instance of [[insert attribute name here]]?" In our Employee data model, the minimal set of primary key instances are **Employee Identifier** and **Department Code.** So these are all valid questions for our employee example:

- Are both **Employee Identifier** and **Department Code** needed to retrieve a single instance of Employee Phone Number?

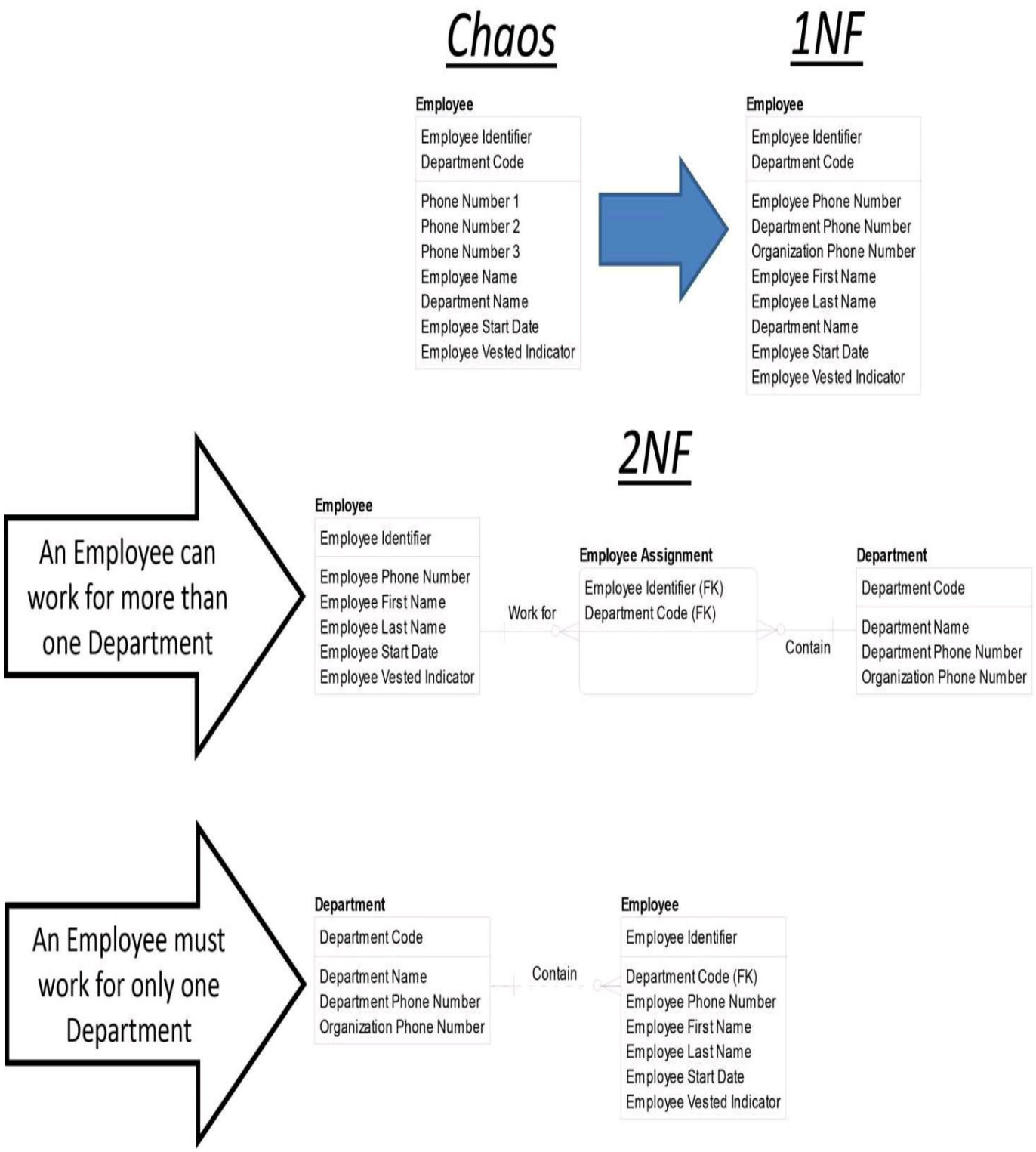
- Are both **Employee Identifier** and **Department Code** needed to retrieve a single instance of **Department Phone Number**?
- Are both **Employee Identifier** and **Department Code** needed to retrieve a single instance of **Organization Phone Number**?
- Are both **Employee Identifier** and **Department Code** needed to retrieve a single instance of **Employee First Name**?
- Are both **Employee Identifier** and **Department Code** needed to retrieve a single instance of **Employee Last Name**?
- Are both **Employee Identifier** and **Department Code** needed to retrieve a single instance of **Department Name**?
- Are both **Employee Identifier** and **Department Code** needed to retrieve a single instance of **Employee Start Date**?
- Are both **Employee Identifier** and **Department Code** needed to retrieve a single instance of **Employee Vested Indicator**?

We learned from asking these questions that **Employee Identifier** is needed to retrieve a single instance of **Employee Phone Number**, **Employee First Name**, **Employee Last Name**, **Employee Start Date**, and **Employee Vested Indicator**. We learned also that **Department Code** is needed to retrieve a single instance of **Department Name**, **Department Phone Number**, and **Organization Phone Number**.

Normalization is a process of asking business questions. In this example, we could not complete 2NF without asking the business "Can an **Employee** work for more than one **Department** at the same time?" If the answer is "Yes" or "Sometimes," then the first model under 2NF on the facing page is accurate. If the answer is "No," then the second model prevails.

Third Normal Form (3NF)

Recall that the series of rules can be summarized as: *Every attribute is single-valued and provides a fact completely and only about its primary key*. First Normal Form (1NF) is the "single-valued" part. Second Normal Form (2NF) is the "completely" part. Third Normal Form (3NF) is the "only" part.



3NF requires the removal of hidden dependencies. A hidden dependency is when a property of an entity depends upon one or more other properties in that same entity instead of directly on that entity's primary key. Each attribute must be directly dependent on only the primary key and not directly dependent on any other attributes within the same entity.

For example, assume an **Order** is identified by an **Order Number**. Within **Order**,

there are many attributes, including **Order Scheduled Delivery Date**, **Order Actual Delivery Date**, and **Order On Time Indicator**. **Order On Time Indicator** contains either a *Yes* or a *No*, providing a fact about whether the **Order Actual Delivery Date** is less than or equal to the **Order Scheduled Delivery Date**. **Order On Time Indicator**, therefore, provides a fact about **Order Actual Delivery Date** and **Order Scheduled Delivery Date**, not directly about **Order Number**. **Order Delivered On Time Indicator** has a hidden dependency therefore, that must be addressed to put the model into 3NF.

The data model is a communication tool. The relational logical data model communicates which attributes are facts about the primary key and only the primary key. Hidden dependencies complicate the model and make it difficult to determine how to retrieve values for each attribute.

To resolve a hidden dependency, you will either need to remove the attribute that is a fact about non-primary key attribute(s) from the model, or you will need to create a new entity with a different primary key for the attribute that is dependent on the non-primary key attribute(s).

As with 1NF and 2NF, we will find ourselves asking many questions to uncover hidden dependencies. We can have another question template such as:

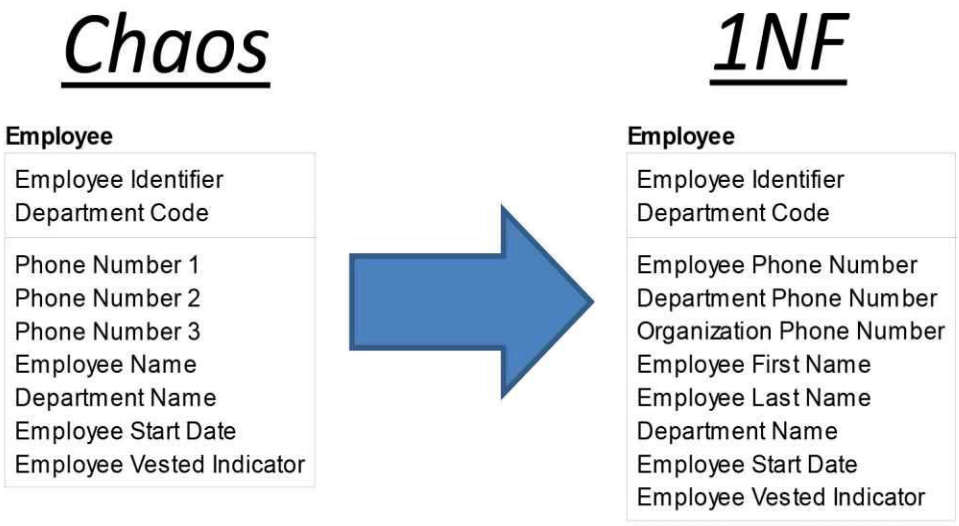
"Is [[insert attribute name here]] a fact about any other attribute in this same entity?"

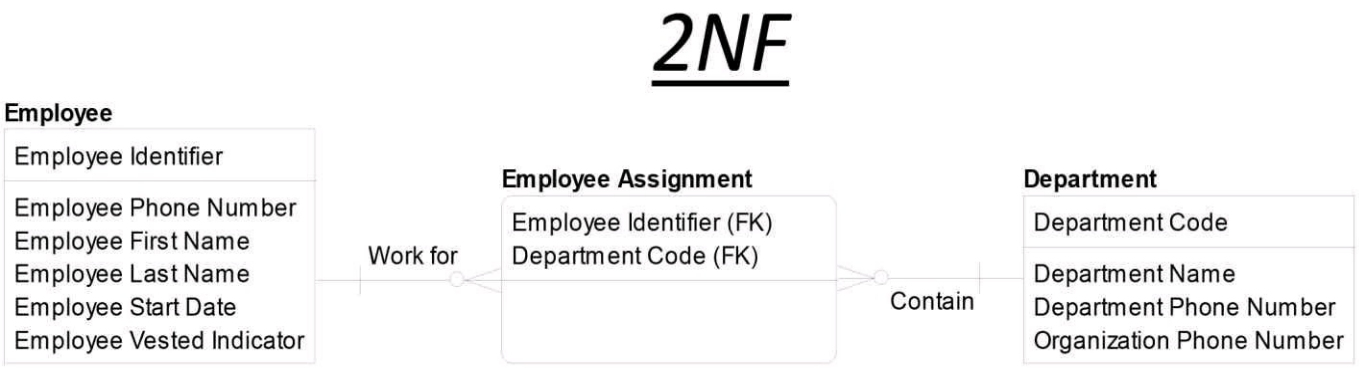
So these are all valid questions for our employee example:

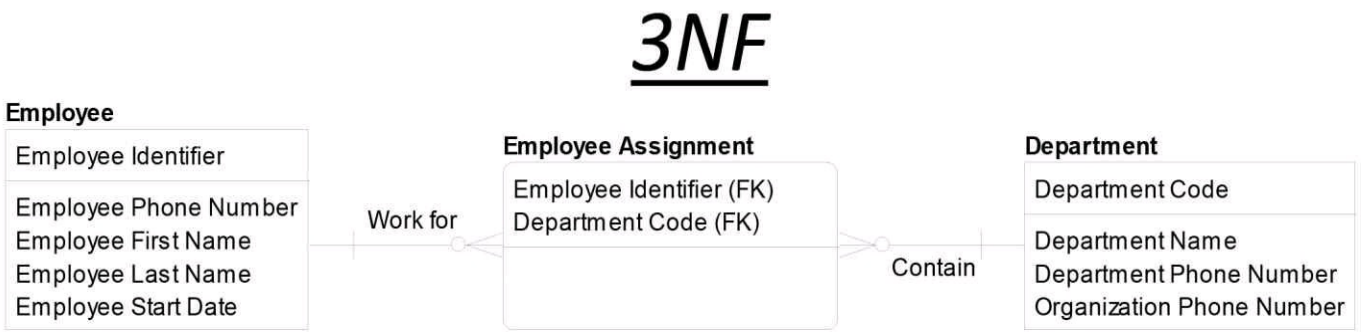
- Is Employee Phone Number a fact about any other attribute in **Employee**?
- Is **Organization Phone Number** a fact about any other attribute in **Department**?
- Is **Department Phone Number** a fact about any other attribute in **Department**?
- Is **Employee First Name** a fact about any other attribute in **Employee**?
- Is **Employee Last Name** a fact about any other attribute in **Employee**?
- Is **Department Name** a fact about any other attribute in **Department**?
- Is **Employee Start Date** a fact about any other attribute in **Employee**?
- Is **Employee Vested Indicator** a fact about any other attribute in **Employee**?

Note that **Employee Vested Indicator** may be a fact about **Employee Start Date** as **Employee Vested Indicator** can be calculated *Y* or *N* based upon the employee's start

date. The following figure shows the model in 3NF after removing the derived attribute **Employee Vested Indicator.**







You will find that the more you normalize, the more you go from applying rules sequentially to applying them in parallel. For example, instead of first applying 1NF to your model everywhere and then, when you are done, applying 2NF and so on, you will find yourself looking to apply all levels at once. This can be done by looking at each entity and making sure the primary key is correct and that it contains a minimal set of attributes and that all attributes are facts about only the primary key.

Abstraction (Relational Only)

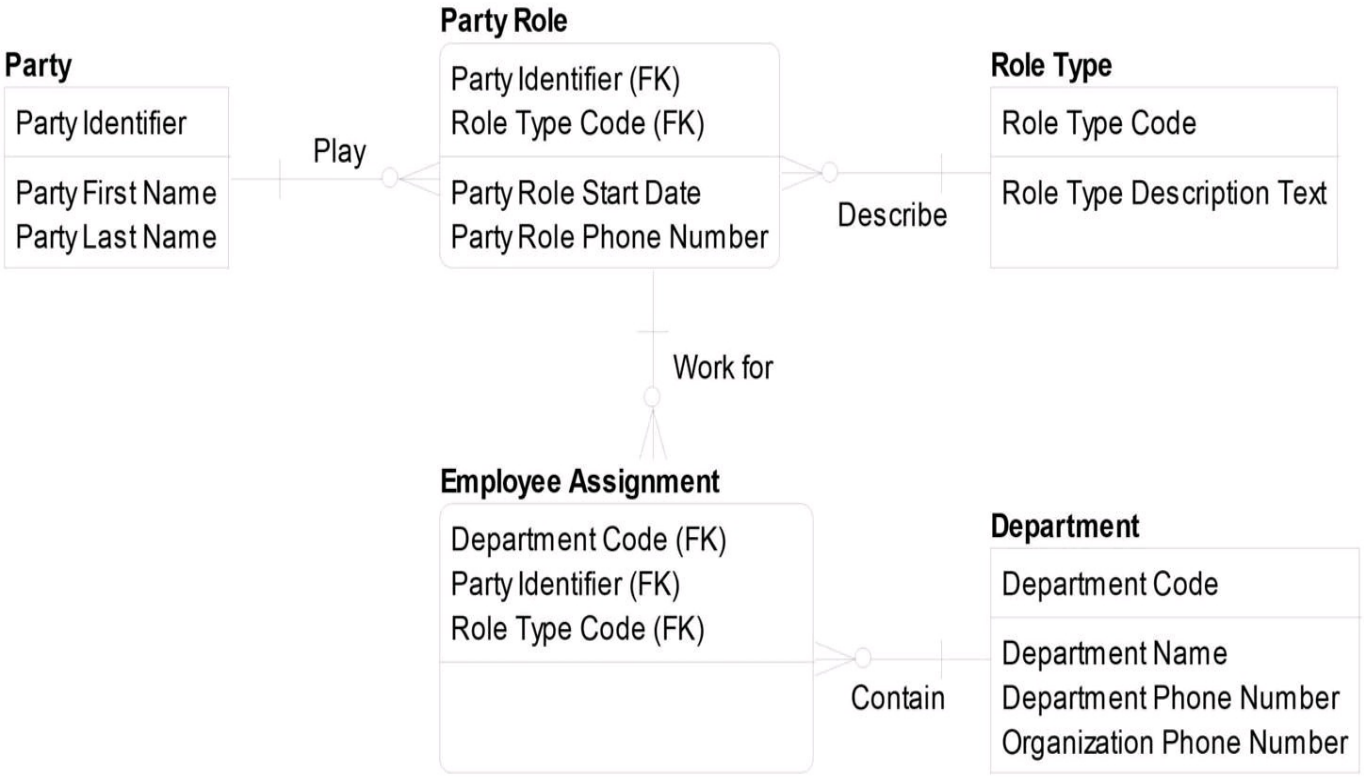
Normalization is a mandatory technique on the relational logical data model.

Abstraction is an optional technique. Abstraction brings flexibility to your data models by redefining and combining some of the attributes, entities, and relationships within the model into more generic terms.

The general question that needs to be asked is:

"Are there any entities, relationships, or data elements that should be made more generic to accommodate future requirements?"

For example, we can take our normalized data model from the prior section and abstract **Employee** into **Party** and **Role** to get this model:



Where did our employees go? They are now a particular role that a **Party** can play. Party can be any person (and usually also any organization, but in this example, only person), and a person can play many roles such as employee, tennis player, or student. Which model do you like better, the 3NF model we built or this abstract model? Most of us like the 3NF model better because the abstract model has several disadvantages:

- **Loss of communication**. The concepts we abstract are no longer represented explicitly on the model. That is, when we abstract, we often convert column names to entity instances. For example, **Employee is no longer an explicit entity**, but is, instead, an entity instance of **Party Role**, with a Role Type Code value of *03* for *Employee*. One of the main reasons we model is for

communication, and abstracting can definitely hinder communication.

- **Loss of business rules**. When we abstract, we can also lose business rules. To be more specific, the rules we enforced on the data model before abstraction now need to be enforced through other means such as through programming code. If we wanted to enforce that an **Employee** must have a **Start Date**, for example**,** we could no longer enforce this rule through our abstracted data model.
- **Additional development complexity**. Abstracting requires sophisticated development techniques to turn attributes into values when loading an abstract structure, or to turn values back into attributes when populating a structure from an abstract source. Imagine the work to populate **Party Role** from the source **Employee**. It would be much easier for a developer to load data from an entity called **Employee** into an entity called **Employee**. The code would be simpler, and it would be very fast to load.

So, although abstraction provides flexibility to an application, it does come with a cost. It makes the most sense to use abstraction when the modeler or analyst anticipates additional *types* of something coming in the near future. For example, in our abstract data model, additional types of people might become important in the future such as **Contractor** and **Consumer**. Roles such as **Contractor** and **Consumer** can be added gracefully without updates to our model. Abstraction might also be used when there is an integration requirement such as the requirement to recognize that *Bob* the employee is also *Bob* the student.

To minimize the cost of abstraction, some modelers prefer to wait until the physical relational data model before abstracting. This way the logical data model still reflects the business language, and then flexibility and integration can be performed in the application through abstraction in the physical data model.

If we are building a MongoDB database, abstraction can be even less desirable because MongoDB does not require us to define our physical schema in advance; therefore, we lose the flexibility benefit of abstraction because MongoDB is already flexible by design. Therefore, the only benefit of abstraction in a MongoDB world is when there is a requirement for integration and the abstract concepts such as **Party** serve as integration points in our application.

STEP 3: DETERMINE THE MOST USEFUL FORM

Recall during conceptual data modeling how we needed to identify the validator (the person or group responsible for ensuring our modeling results are correct) and the users (the person or group who is going to use the data model). We need to perform the same activity at the logical level. Determine who the validators and users will be

at the logical level, and that will determine the most useful form.

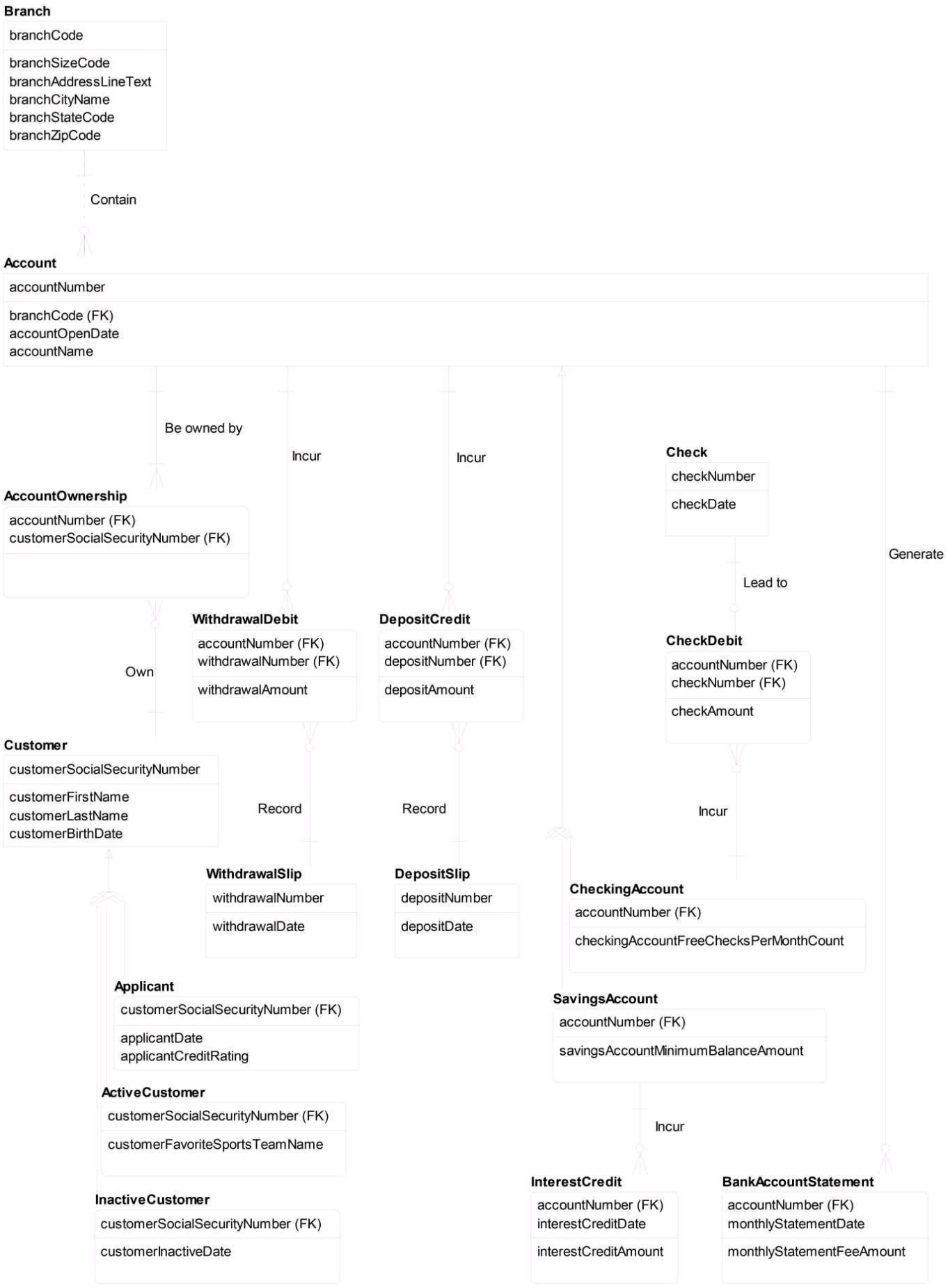
At the conceptual level, it is very possible for the validators and users to have different technical levels and experiences; therefore, there could be more than one form for communicating the conceptual data model. At the logical level, however, you will find a majority of the time that the same form will work well for both the validators and users.

Let's see the different forms we can choose from for both relational and dimensional perspectives.

For Relational

At the relational logical level, we do have the option of multiple forms. However, it would be very difficult (but possible) to represent the LDM using business assertions or a business sketch, both of which work well on the CDM. Variations on the relational LDM have more to do with notation, such as using Information Engineering (IE for short), which is the notation in this book, versus using the UML Class Diagram, for example.

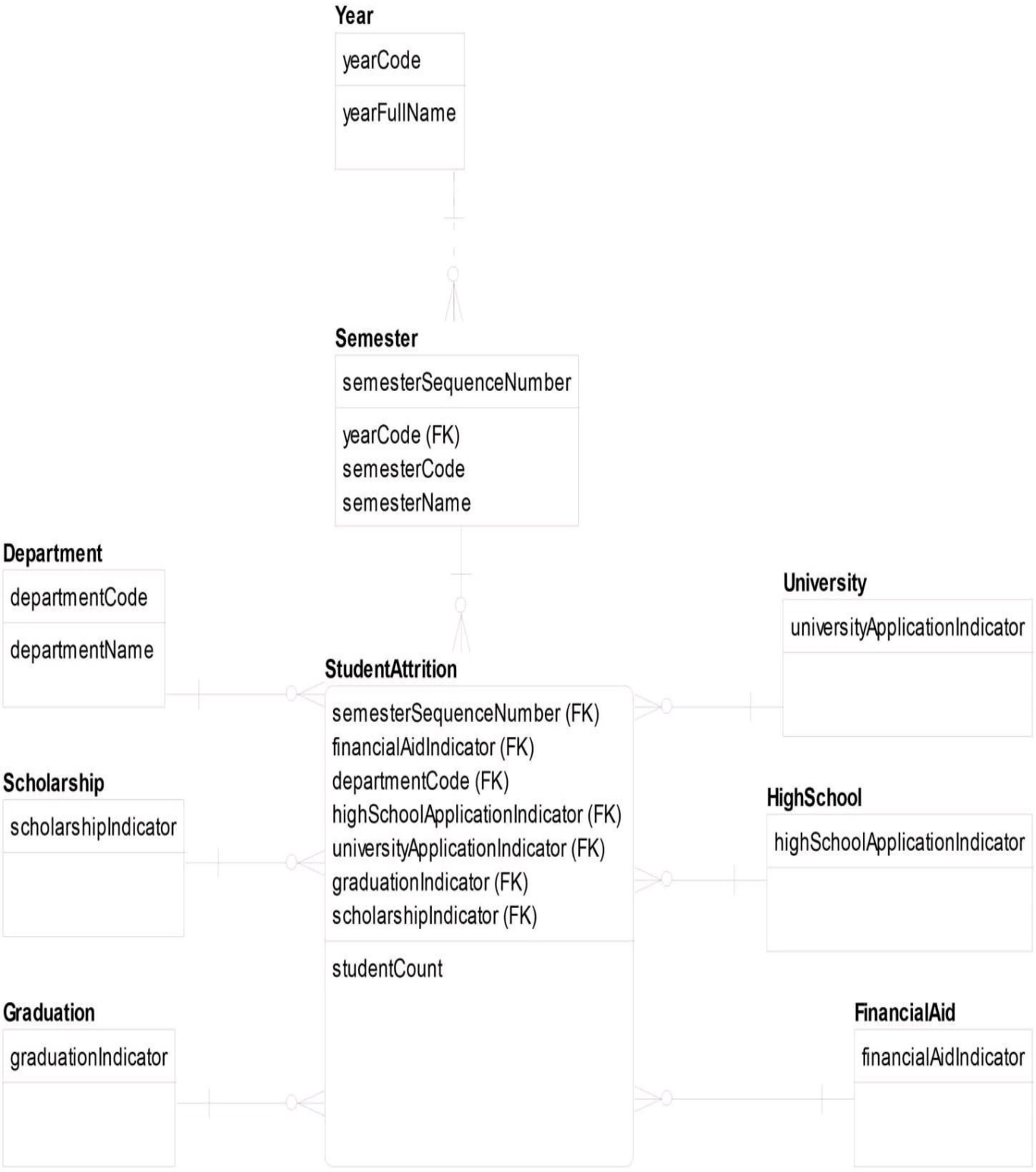
In our Account example, here is our relational LDM after normalizing, using the IE notation:


Branch
branchCode
branchSizeCode
branchAddressLineText
branchCityName
branchStateCode
branchZipCode
Contain
Account
accountNumber
branchCode (FK)
accountOpenDate
accountName
Be owned by
Incur
Incur
Generate
Check
checkNumber
checkDate
Lead to
AccountOwnership
accountNumber (FK)
customerSocialSecurityNumber (FK)
Own
WithdrawalDebit
accountNumber (FK)
withdrawalNumber (FK)
withdrawalAmount
DepositCredit
accountNumber (FK)
depositNumber (FK)
depositAmount
CheckDebit
accountNumber (FK)
checkNumber (FK)
checkAmount
Customer
customerSocialSecurityNumber
customerFirstName
customerLastName
customerBirthDate
Record
Record
Incur
WithdrawalSlip
withdrawalNumber
withdrawalDate
DepositSlip
depositNumber
depositDate
CheckingAccount
accountNumber (FK)
checkingAccountFreeChecksPerMonthCount
Applicant
customerSocialSecurityNumber (FK)
applicantDate
applicantCreditRating
SavingsAccount
accountNumber (FK)
savingsAccountMinimumBalanceAmount
ActiveCustomer
customerSocialSecurityNumber (FK)
customerFavoriteSportsTeamName
Incur
InactiveCustomer
customerSocialSecurityNumber (FK)
customerInactiveDate
InterestCredit
accountNumber (FK)
interestCreditDate
interestCreditAmount
BankAccountStatement
accountNumber (FK)
monthlyStatementDate
monthlyStatementFeeAmount

Note that often during the process of building the logical data model, we refine (and sometimes correct) rules we identified during the conceptual data modeling stage. For example, upon examining the primary key to **MonthlyStatementFee**, we learned that it shared the same primary key as **BankAccountStatement**. After further analysis, we moved the fee amount to **BankAccountStatement** and removed **MonthlyStatementFee**. We also learned that there is a one-to-many relationship between **Account** and **BankAccountStatement**, not a many-to-many relationship. We also uncovered the new property, **monthlyStatementDate**, which is needed to distinguish **BankAccountStatements** belonging to the same **Account**.

For Dimensional

Similar to relational, there are multiple notations we can choose for the dimensional logical. Here is our University dimensional LDM using the traditional information engineering notation:



Recall from our conceptual data modeling discussion that **Student Attrition** is an example of a fact table (on a conceptual and logical data model often called a "meter"). A meter is an entity containing a related set of measures. The measures on this model determine how the student attrition business process is doing—in this case, **studentCount**. Often the meter contains many measures. Also, often the dimensions

contain many attributes, sometimes hundreds of them. On this model we have a number of dimensions, such as **University**, that only contain a single attribute.

STEP 4: REVIEW AND CONFIRM

Previously we identified the person or group responsible for validating the model. Now we need to show them the model and make sure it is correct. Often at this stage after reviewing the model we go back to Step 1 and make some changes and then show the model again. This iterative cycle continues until the model is agreed upon by the validator and deemed correct so that we can move on to the physical modeling phase.

EXERCISE 6: LOGICAL DATA MODELING MINDSET

Recall the MongoDB document below, which we discussed in our last chapter. Think of at least ten questions that you would ask during the logical data modeling phase based upon the data you see in this collection. See Appendix A for some of the questions I would ask.

Order:

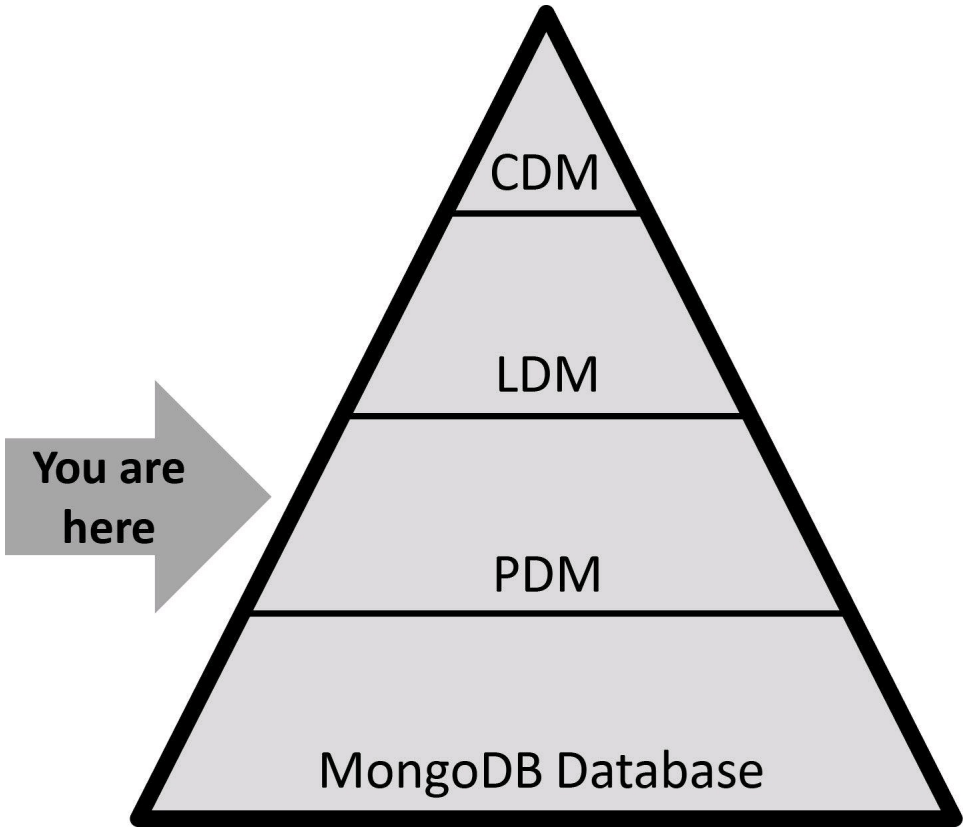
```
{  orderNumber : “4839-02”,
                    orderShortDescription : “Professor review copies of several titles”,
   orderScheduledDeliveryDate : ISODate(“2014-05-15”),
   orderActualDeliveryDate : ISODate(“2014-05-17”),
   orderWeight : 8.5,
   orderTotalAmount : 19.85,
   orderTypeCode : “02”,
   orderTypeDescription : “Universities Sales”,
   orderStatusCode : “D”,
   orderStatusDescription : “Delivered”,
   orderLine :
    [ {  productID : “9781935504375”,
      orderLineQuantity : 1
    },
                                                    {    productID : “9781935504511”,
      orderLineQuantity : 3
    },
    {  productID : “9781935504535”,
      orderLineQuantity : 2
```

```
} ] }
```

Key Points

- Logical data modeling is the process of capturing the detailed business solution. The logical data model looks the same regardless of whether we are implementing in MongoDB or Oracle.
- There are four steps to logical data modeling: Fill in the CDM (Step 1), normalize and abstract for relational (Step 2), determine the most useful form (Step 3), and review and confirm (Step 4).
- A relational logical data model represents the detailed workings of the business. A dimensional logical data model represents the details necessary to answer one or more business questions.
- Normalizing is a formal process of asking business questions. Normalization ensures that every attribute is a fact about the key (1NF), the whole key (2NF), and nothing but the key (3NF).
- Abstraction is the use of generic concepts such as **Party**. Abstraction has as advantages flexibility and integration, but it comes with the high price of obscurity in the form of loss of communication, loss of business rules, and added complexity.
- MongoDB does not require us to define our physical schema in advance; therefore, we lose the flexibility benefit of abstraction because MongoDB is already flexible by design.

Chapter 7
Physical Data Modeling



Physical data modeling is the process of capturing the detailed technical solution. This is the first time we are actually concerning ourselves with technology and technology issues such as performance, storage, and security. For example, conceptual data modeling captures the satellite view of the business requirements, revealing that a **Customer** places many **Orders**. Logical data modeling captures the detailed business solution, which includes all of the properties of **Customer** and **Order** such as the customer's name, their address, and the order number. After understanding both the

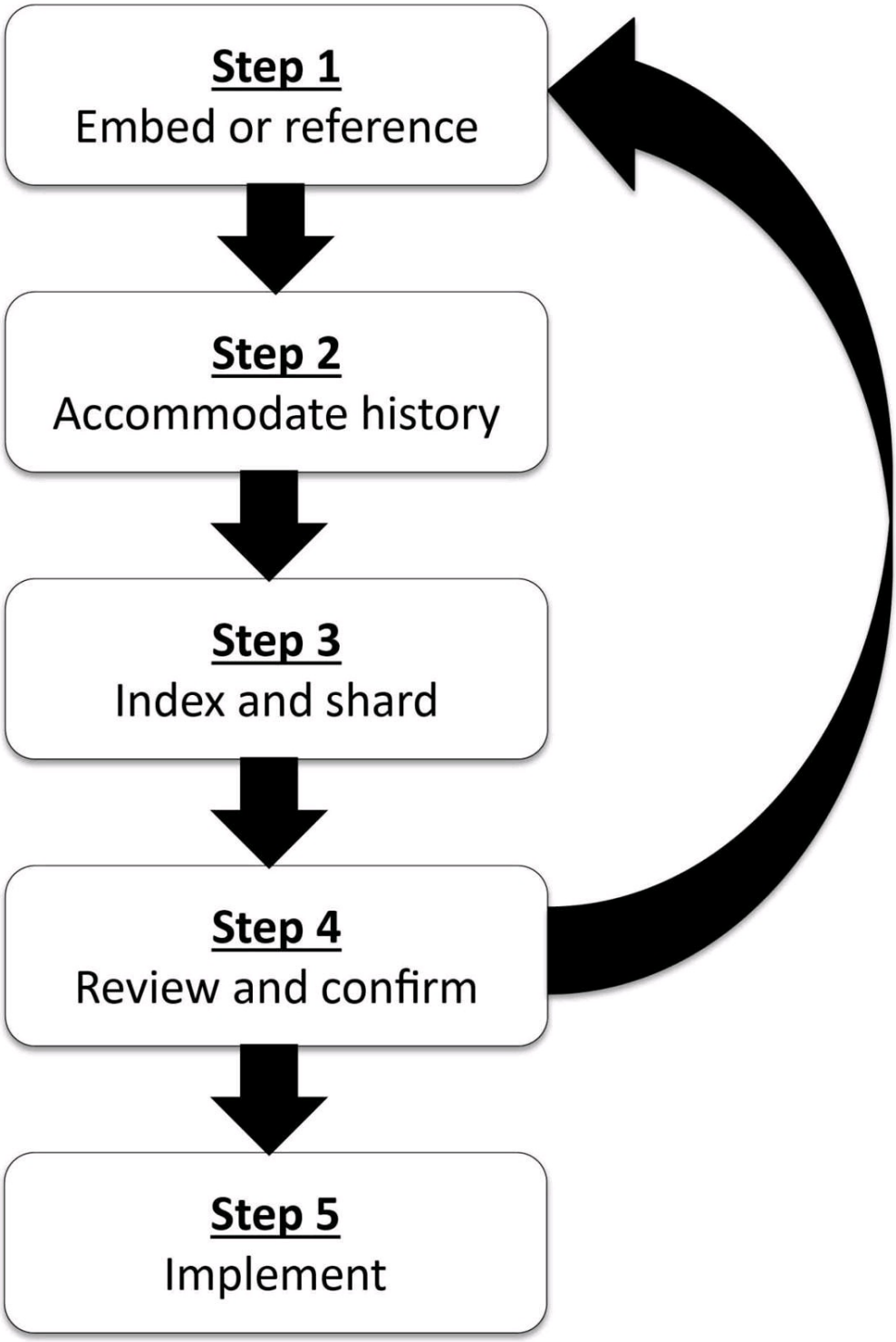
high level and detailed business solution, we move on to the technical solution, and physical data modeling may lead to embedding **Order** within **Customer** in MongoDB.

In physical data modeling, we aim to build an efficient MongoDB design, addressing questions such as:

- What should the collections look like?
- What is the optimal way to do sharding?
- How can we make this information secure?
- How should we store history?
- How can we answer this business question in less than 300 milliseconds?

Physical Data Modeling Approach

There are five steps to physical data modeling:



Starting with our logical data model, we prepare the initial MongoDB structure by deciding for each relationship whether to embed or reference (Step 1). After making

these changes to the model, we can determine where to accommodate history (Step 2). History means keeping track of how things change over time. There most likely will be other structural refinement needed such as adding indexes or sharding; both techniques are used to further improve retrieval performance (Step 3). In Step 4, review the work to get approval. Frequently, action items come out of the review and confirm step where we need go back to Step 1 and refine the structure. Expect iteration through these steps. Once we have received final confirmation from our work, we can move on to implementation. Let's talk more about each of these five steps.

STEP 1: EMBED OR REFERENCE

MongoDB resolves relationships in one of two ways: embed or reference. Recall from Chapter 3 that embedded documents resolve relationships between data by storing related data in a single document structure, and a reference is a pointer to another document. MongoDB documents make it possible to embed document structures as sub-documents in a field or an array within a document. Embed is equivalent to the concept of denormalization, in which entities are combined into one structure with the goal of creating a more simplistic and performant structure, usually at the cost of extra redundancy and, therefore, storage space. One structure is often easier than five structures from which to retrieve data. Also, the one structure can represent a logical concept such as a survey or invoice or claim.

If I need to retrieve data, it can (most of the time) take more time to get the data out of multiple documents by referencing than by embedding everything into a single document. That is, having everything in one document is often better for retrieval performance than referencing. If I frequently query on both **Order** and **Product**, having them in one document would most likely take less time to view than having them in separate documents and having the **Order** reference the **Product**.

Notice the language though in the preceding paragraph; "most of the time" and "most likely" imply that embedding is not always faster, nor should retrieval speed be the only factor to consider when deciding whether to embed or reference. The top five reasons for embedding over referencing are:

- **Requirements state that data from two or more entities are frequently queried together**. If we are often viewing data from multiple entities together, it can make sense to put them into one document. It is important to note that this factor, which is possibly the most important factor in deciding whether to embed or reference, has more to do with usage than technology. No matter how skilled we are with MongoDB, we still need to know the business requirements!

2. **The child is a dependent entity**. In the following model, **Customer** and **Account** in Example 1, and **Customer** in the Example 2 are independent entities (shown as rectangles), and **Account** in Example 2 is a dependent entity (shown as a rectangle with rounded corners).

Example 1:

Example 2:

An independent entity is an entity where each occurrence (instance) can be found using only its own attributes such as a **Customer ID** for **Customer**. A dependent entity, such as **Account** in the second example, contains instances that can only be found by using at least one attribute from a *different* entity such as **Customer ID** from **Customer**.

To make the explanation of independent and dependent clearer, here are our two data models with example values for two attributes within **Account**, **Account Number** and the foreign key back to **Customer**, **Customer ID**.

Example 1:

Customer ID	Account Number
123	34
123	37
156	42
167	16

Example 2:

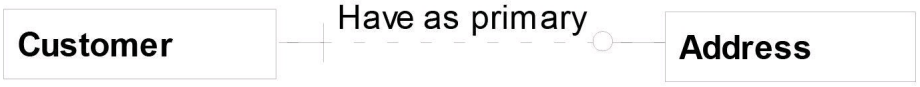
Customer ID	Account Number
123	34
123	37
156	34
167	34

On the model in Example 1, we see that **Account Number** is unique, and therefore to retrieve a specific **Account** requires only knowing the **Account Number** and not anything from **Customer**, making **Account** an independent entity because **Account Number** belongs to **Account**. However, in Example 2, **Account Number** *34* appears three times, so the only way to find a specific account is to know who the customer is. So the combination of **Customer ID** and **Account Number** is required to distinguish a particular account, making **Account** in Example 2 a dependent entity because **Customer ID**, which is needed to help identify accounts, does not belong to **Account**.

Notice also that the relationship line looks different when connecting to an independent verses dependent entity. The dotted line means *non-identifying* and the solid line means *identifying*. Identifying relationships mean that the entity on the many side (the child) is always going to be a dependent entity (like Account in Example 2) to the entity on the one side (the parent).

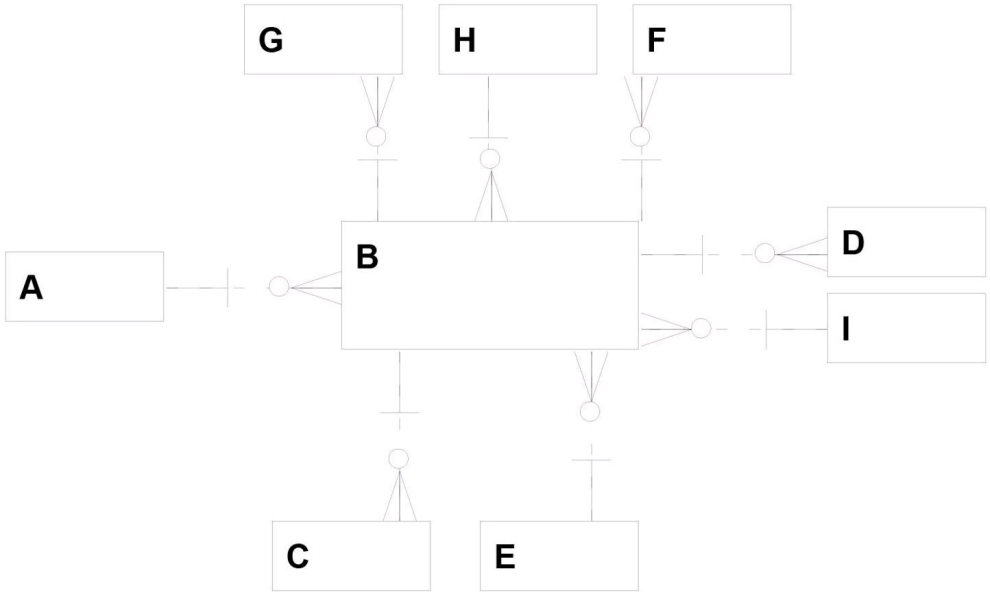
If we are designing a MongoDB database and trying to decide whether we should embed documents or reference other documents, a major factor would be whether the relationship is identifying or non-identifying. An identifying relationship is a much stronger relationship than non-identifying, giving us more weight towards embedding the weak entity within the strong entity so that we have just one document.

There is a one-to-one relationship between two entities. A one-to-one relationship means that for a given entity instance, there can be at most one entity instance relating to it, meaning if we decide to embed, there will be no redundancies, thus creating a simple easy-to-query structure. In this model, we would embed **Address** into **Customer**:



Similar volatility. If the entity you are considering embedding within another entity experiences updates, inserts, and deletes at a similar rate to the entity it will be embedded into, there is a stronger reason to combine them. The opposite is the argument for referencing. That is, if you are considering embedding **Customer Contact** into **Customer** and **Customer Contact** information experiences almost hourly changes while **Customer** is relatively stable, there is a greater tendency towards referencing **Customer** from **Customer Contact** instead of embedding. This volatility factor also includes considering chronology requirements. That is, if there is a requirement to store all history (e.g. auditing) on one entity but a current view only on the other entity, you may lean towards referencing over embedding.

If the entity you are considering embedding is not a key entity. If **Claim** and **Member** have a relationship but **Claim** has many other relationships, it may be prudent to reference **Member** from **Claim** instead of embedding the key entity **Claim** into **Member**. It is more efficient and less error prone to reference the entity that is needed by many other entities. Another example are many-to-many relationships where the associative entity (the entity that resolves the many-to-many), is referenced by at least two other entities. In these situations it may be better to reference rather than embed. For example, I would not consider embedding entity **B** into any of the other entities on this model (I might consider embedding one or a few of the surrounding entities into **B**, however):



Two more points on embedding versus referencing:

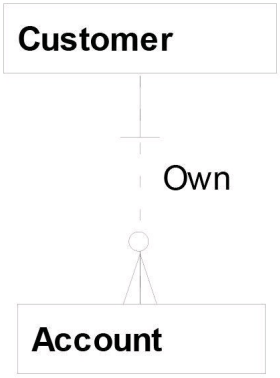
- If some of these five factors apply but others don't, the best option is to test both ways (embed and reference) and go with the one that gives the best results, where results include retrieval performance, ease of data updates, and storage efficiencies. Because we are working without predefined schemas, it is a lot easier to test in MongoDB than in a relational database environment.
- MongoDB imposes a size limit on a single document of 16 MB. It is very unlikely the size factor will influence your decision as to whether to embed or reference, as this is a huge amount of space (e.g. 120 million tweets), and therefore this factor rarely becomes a decision point. In addition, MongoDB arrays that grow very large tend to perform poorly even when indexed. If a set of values or sub-documents will grow significantly over time, consider referencing over embedding to avoid this performance penalty.[5]

EXERCISE 7: Embed or Reference

Below are two scenarios with accompanying data models. Based on the scenario and the sample MongoDB document, decide whether you would embed or reference. Please see Appendix A for my responses.

Scenario	Sample Embedded Document	Sample Referenced Document

We are building an application that allows easy querying of customers and their accounts. **Customer** has lots of other relationships, and in this scenario **Account** has no other relationships.



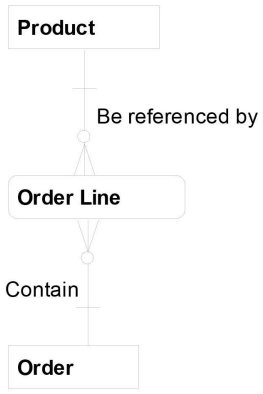

```
Customer : {
  _id : "bob",
  customerName : "Bob Jones",
  customerDateOfBirth : ISODate("1973-10-07"),
accounts : [
{
accountCode : "3281",
  accountOpenDate : ISODate("2014-02-05")
},
{
  accountCode : "12341",
  accountOpenDate : ISODate("2014-03-21")
 } ] }
```

```
Customer : {
  _id : "bob",
  customerName : "Bob Jones",
  customerDateOfBirth : ISODate("1973-10-07")
 }
Account : {
  customerId : "bob",
 accountCode : "3281",
  accountOpenDate : ISODate("2014-02-05")
 }
Account : {
  customerId : "bob",
accountCode : "12341",
  accountOpenDate : ISODate("2014-03-21")
 }
```

We are building an application to allow efficient data entry of product order information. **Order** contains over 100 fields, **Order Line** contains over 50 fields, and **Product** contains over 150 fields. An **Order**, on average, contains 20 **Order Lines**. Would you embed **Order Line** into **Order**?

```
Order : {
  _id : "1234",
  orderNumber : "AB-3429-01",
  orderDate : ISODate("2013-11-08")
},
orderLines : [
 { orderLineSequenceNumber : "1",
productId : "2837",
orderLineQuantity : 15
},
```

```
Order : {
  _id : "1234",
  orderNumber : "AB-3429-01",
 ISODate("2013-11-08")
 }
OrderLine : {
  orderId : "1234",
  orderLineSequenceNumber : "1",
 productId : "2837",
 orderLineQuantity : 15
 }
OrderLine : {
```



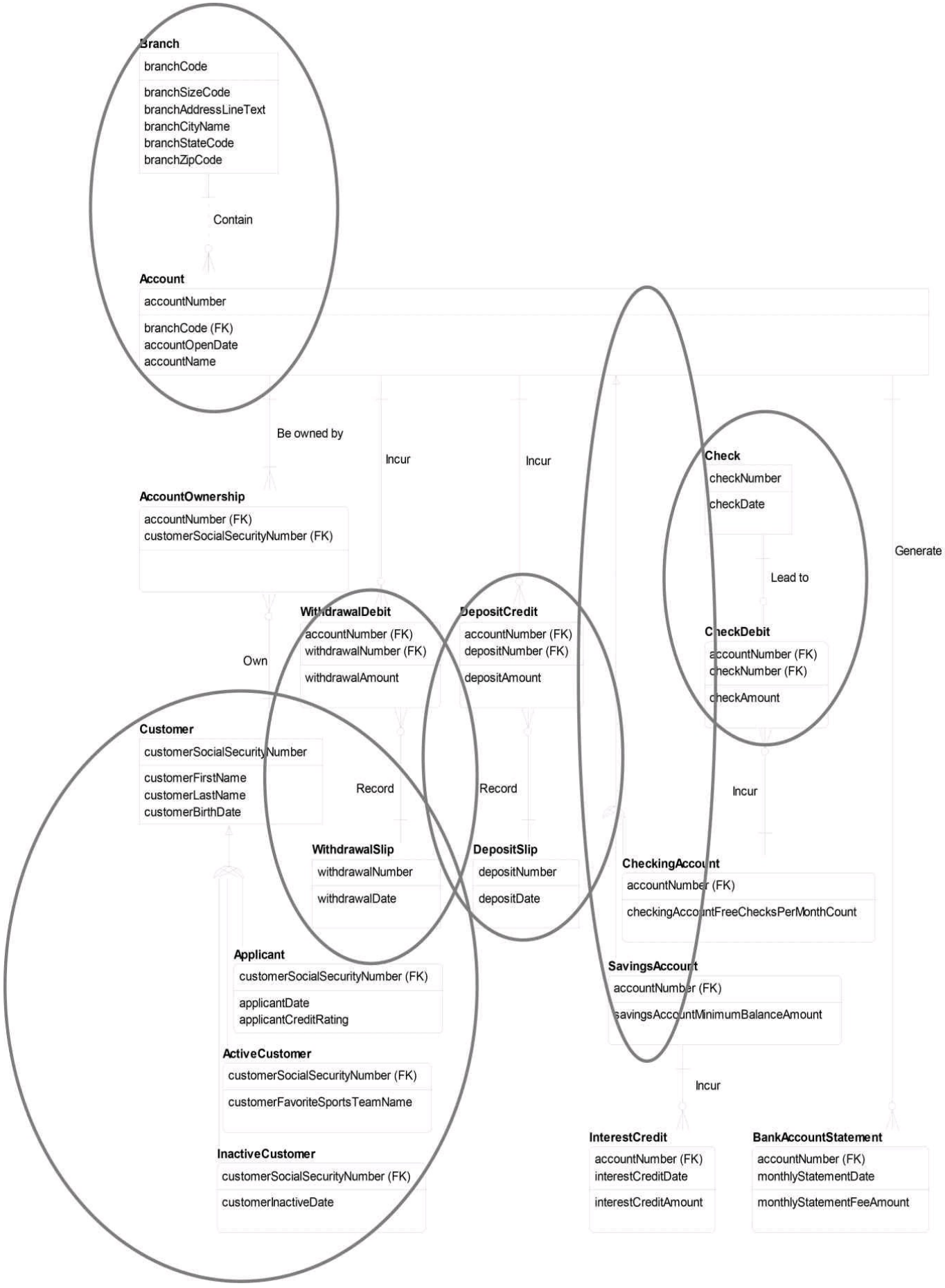

```
{
orderLineSequenceNumber : "2",
productId : "2842",
orderLineQuantity : 25
} ] }
```

```
orderId : "1234",
orderLineSequenceNumber : "2",
productId : "2842",
orderLineQuantity : 25
}
```

Recall from this section's introduction that relational data modeling is the process of capturing how the business *works* by precisely representing business rules, while dimensional data modeling is the process of capturing how the business is *monitored* by precisely representing business questions. The thought process for this step is the same for both relational and dimensional physical data modeling, so let's look at an example of each.

For Relational

On the facing page is our relational logical data model from Chapter 6. I would take a pencil and start grouping the logical entities together that I think would make efficient collections.


Branch
branchCode
branchSizeCode
branchAddressLineText
branchCityName
branchStateCode
branchZipCode
Contain
Account
accountNumber
branchCode (FK)
accountOpenDate
accountName
Be owned by
Incur
Incur
Generate
Check
checkNumber
checkDate
Lead to
AccountOwnership
accountNumber (FK)
customerSocialSecurityNumber (FK)
Own
WithdrawalDebit
accountNumber (FK)
withdrawalNumber (FK)
withdrawalAmount
DepositCredit
accountNumber (FK)
depositNumber (FK)
depositAmount
CheckDebit
accountNumber (FK)
checkNumber (FK)
checkAmount
Customer
customerSocialSecurityNumber
customerFirstName
customerLastName
customerBirthDate
Record
Record
Incur
WithdrawalSlip
withdrawalNumber
withdrawalDate
DepositSlip
depositNumber
depositDate
CheckingAccount
accountNumber (FK)
checkingAccountFreeChecksPerMonthCount
Applicant
customerSocialSecurityNumber (FK)
applicantDate
applicantCreditRating
SavingsAccount
accountNumber (FK)
savingsAccountMinimumBalanceAmount
ActiveCustomer
customerSocialSecurityNumber (FK)
customerFavoriteSportsTeamName
Incur
InactiveCustomer
customerSocialSecurityNumber (FK)
customerInactiveDate
InterestCredit
accountNumber (FK)
interestCreditDate
interestCreditAmount
BankAccountStatement
accountNumber (FK)
monthlyStatementDate
monthlyStatementFeeAmount

Here are my thoughts during this grouping process:

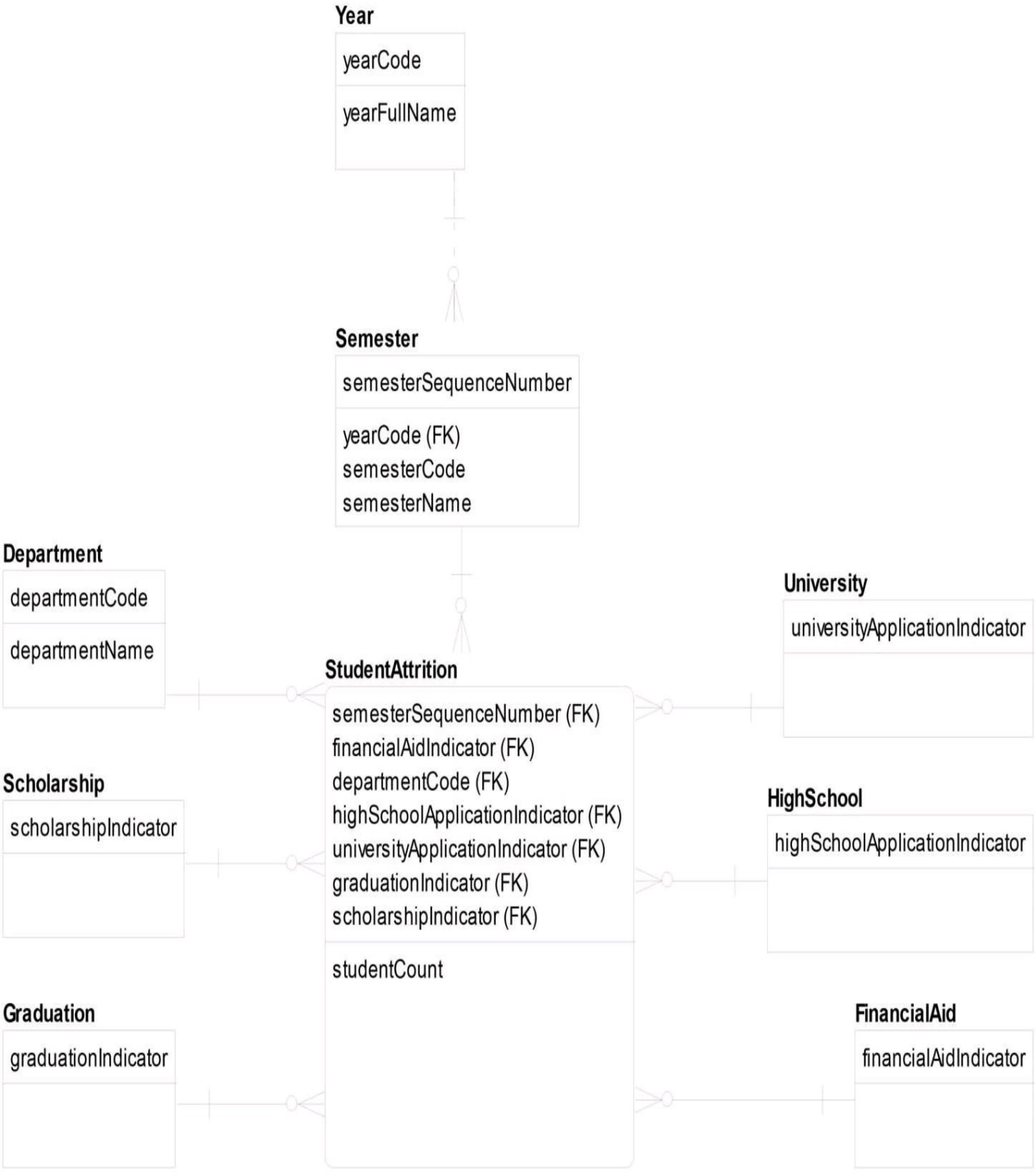
- The subtypes are dependent entities to their supertype, and therefore frequently we roll subtypes up to supertypes and add type codes to the supertype such as **customerTypeCode** and **accountTypeCode**.
- We have a one-to-one relationship between **Check** and **CheckDebit**, which is a great candidate, therefore, for embedding **Check** into **CheckDebit**.
- The documents, such as **WithdrawalSlip**, and their corresponding transactions, such as **WithdrawalDebit**, have similar volatility and therefore are good candidates for embedding. I would not initially embed these transactions and documents into **Account** because of the high difference in volatility with **Account**. We might have millions of transactions, yet only 1,000 accounts that change on a regular basis, for example.
- Assuming stable branch data, we can embed **Branch** in **Account**.
- For now, **Customer** and **Account** remain separate collections. Often many-to-many relationships become references instead of embedded collections because of the complexities of storing a complex relationship in one structure.

Here is the completed relational PDM after making these changes:

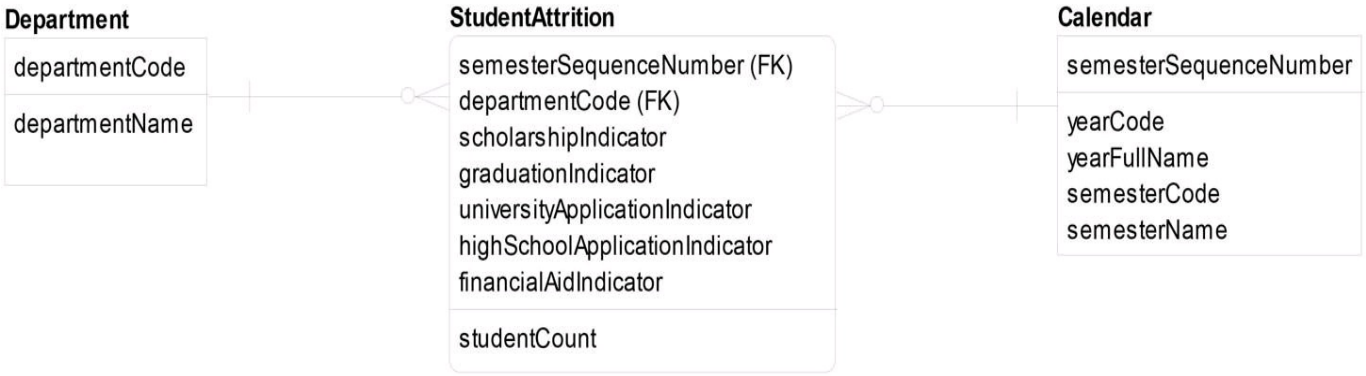
For Dimensional

A star schema is the most common physical dimensional data model structure. The term *meter* from the dimensional logical data model, is replaced with the term *fact table* on the dimensional physical data model. A star schema results when each set of tables that make up a dimension is flattened into a single table. The fact table is in the center of the model, and each of the dimensions relate to the fact table at the lowest level of detail.

Recall the dimensional logical data model example from the last chapter:



Let's convert this to a star schema:



Notice I used a physical dimensional modeling technique called a degenerate dimension in this example. A degenerate dimension is when you remove the dimensional structure and just store the dimensional attribute directly in the fact table. Most of the time, the degenerate dimension contains just a single field—in this case, an indicator. So we removed those dimensions such as **University** that contained only a single field.

STEP 2: ACCOMMODATE HISTORY

There are four options in addressing how field values change over time. Each option for handling these changes is given a different number (either zero, one, two, or three) and is often called a Slowly Changing Dimension (SCD for short). An SCD of Type 0 means we are only storing the original state and not storing changes. That is, a collection of Type 0 means that if the data changes, we just store the way the data was originally and do not apply any changes to it. A Type 1 means the most current view. With a Type 1 we are just storing the most current state of the collection by applying the latest changes. A Type 2 means we are storing all changes to the data in a collection—a complete history whenever anything changes. Type 2 is also known as "auditing" because we are storing all changes and therefore have the ability to analyze the way the world existed at any point in time. Type 2 is like having a time machine. Type 3 means we have a requirement for some history—for example, the most current view and the previous view or the most current view and the original view.

We need to determine the type of history needed for each collection by completing the Collection History Template:

Collection History Template

Store only the original state	Store only the most current state	Store full history	Store some history
(Type 0)	(Type 1)	(Type 2)	(Type 3)

Collection A

Collection B

Collection C

Note that it is not uncommon for a collection to have more than type of history requirement such as **Customer** requiring a Type 1 for **customerLastName** and a Type 2 for **emailAddress**. Also, the thought process for this step is the same for both relational and dimensional physical data modeling, so let's look at an example of each.

For Relational

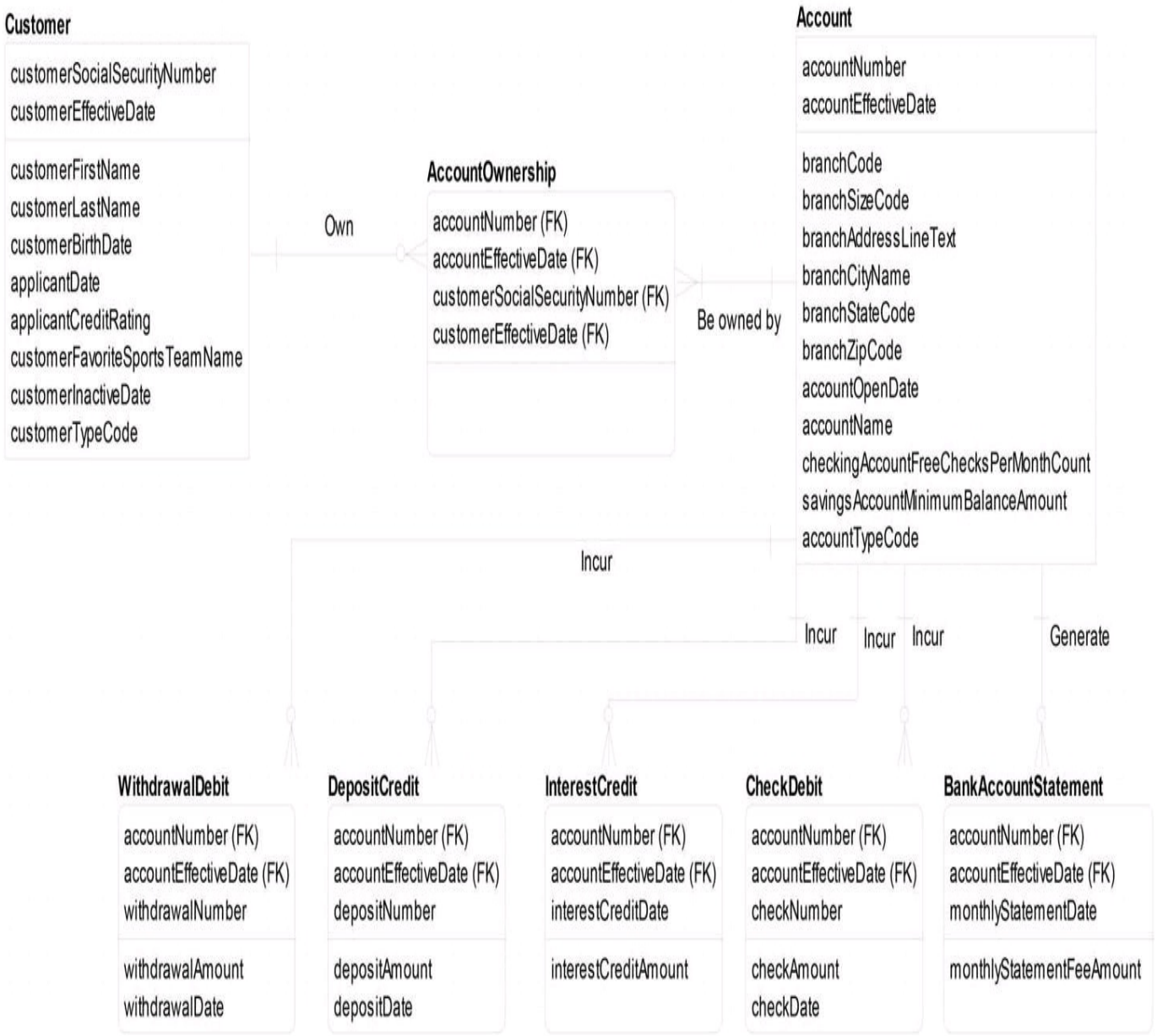
Here is the completed Collection History Template for our Account example:

Account Project Collection History

	Store only the original state (Type 0)	Store only the most current state (Type 1)	Store full history (Type 2)	Store some history (Type 3)
Customer			✓	
AccountOwnership			✓	
Account			✓	
WithdrawalDebit		✓		

DepositCredit	✓
InterestCredit	✓
CheckDebit	✓
BankAccountStatement	✓

Note that transactions and documents are often created but rarely change, which is why a Type 1 is chosen. Note also that it is common to have a Type 2 on **Customer** and **Account**. There are a number of ways to model a Type 2, the most common being to put an effective date in the entity's primary key:



There are other ways of modeling a Type 2, some of which are more efficient for child entities, such as **WithdrawalDebit**, which now has a larger and more complex primary key.

For Dimensional

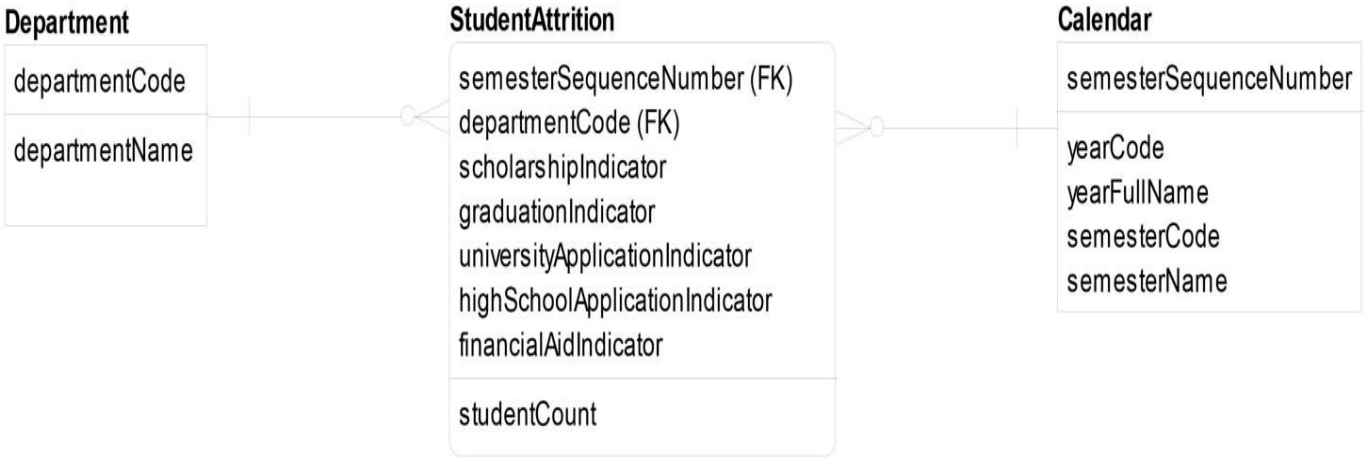
Here is the completed Collection History Template for our University example:

Student Attrition Collection History

	Store only the original state	**Store only the most current state**	**Store full history**	**Store some history**
	(Type 0)	**(Type 1)**	**(Type 2)**	**(Type 3)**

Department	✓
Calendar	✓

Note that because Type 1 is requested for both **Department** and **Calendar**, our data model does not need to change and remains with the same structure as our prior step:



STEP 3: INDEX AND SHARD

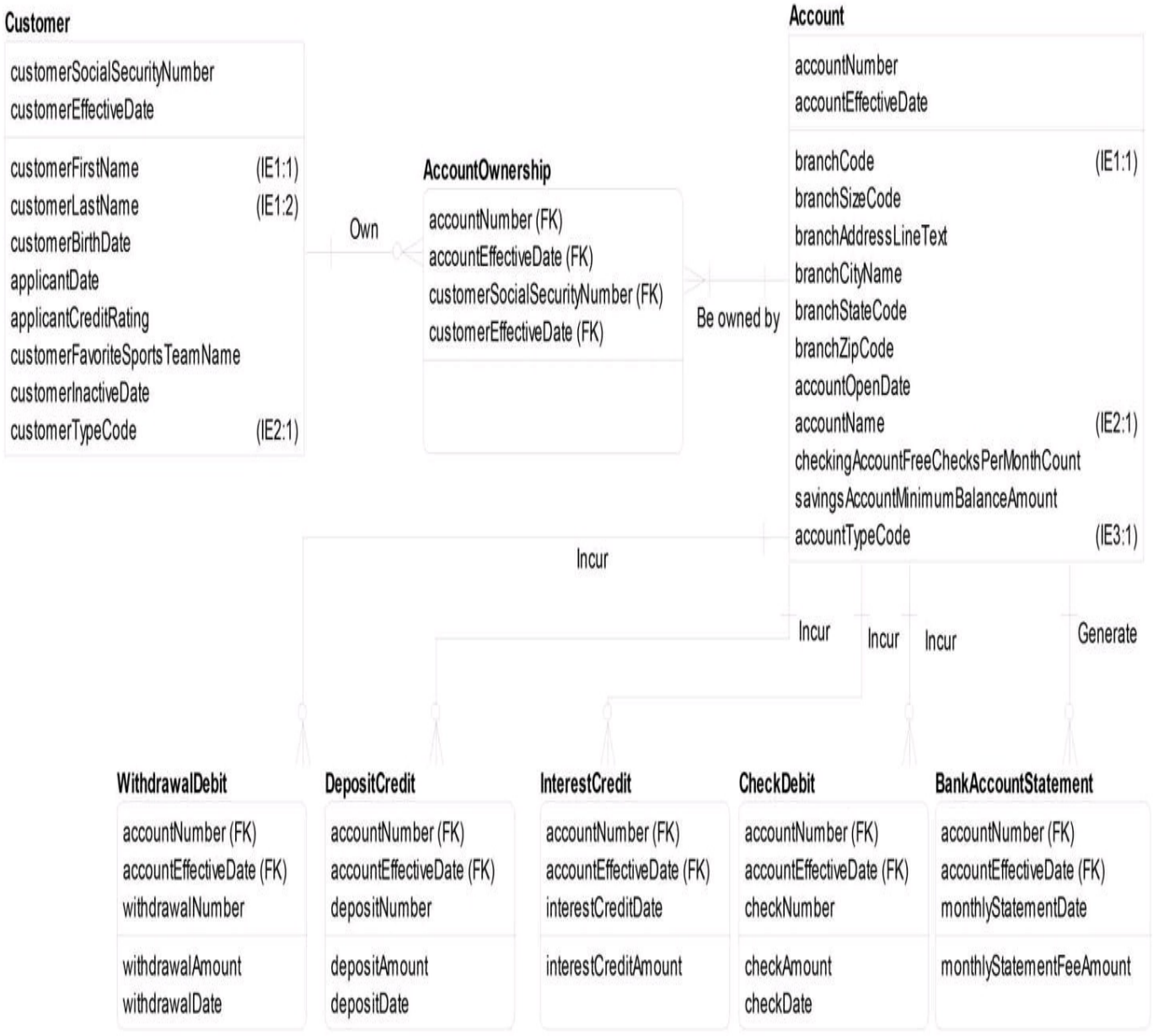
The primary and alternate keys from our logical data model are converted into unique indexes on the physical. We can also add additional indexes for performance, called secondary or non-unique indexes. For example, if there is a continuous need to see **customerLastName** and **customerFirstName**, we can add indexes to these fields to reduce the amount of time it takes to retrieve.

Partitioning (called "sharding" in MongoDB) is when a document is split up into two or more parts. Vertical partitioning is when fields are split up and horizontal is when documents are split up. Frequently, both horizontal and vertical are used together. That is, when splitting documents apart, in many cases we learn that certain fields only belong with certain documents.

Both vertical and horizontal partitioning are common techniques when building analytics systems. A collection might contain a large number of fields and perhaps only a subset are volatile and change often, so this subset can be vertically partitioned into a separate collection. Or we might have ten years of orders in a table, so to improve query performance we horizontally partition by year so that when queries are run within a given year, the performance will be much faster.

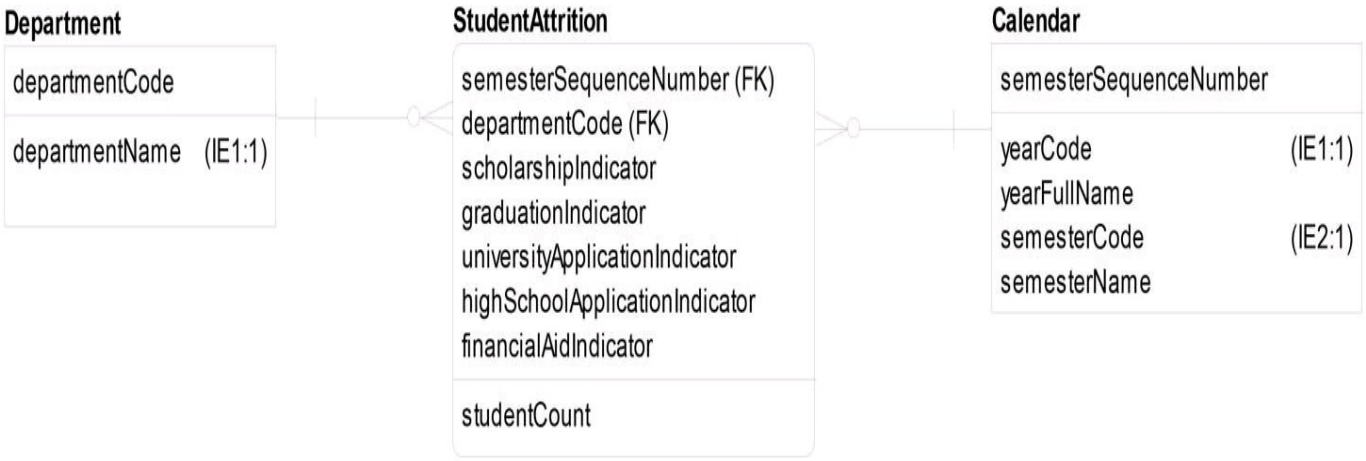
For Relational

For our relational physical data model, these attributes are queried frequently and therefore need non-unique indexes: **customerFirstName** and **customerLastName** (we added a composite index[6] on both of these attributes), **customerTypeCode**, **accountTypeCode**, **branchCode**, and **accountName**:



For Dimensional

For our dimensional physical data model, these attributes are queried frequently and therefore need non-unique indexes: **departmentName**, **yearCode**, and **semesterCode**:



STEP 4: REVIEW AND CONFIRM

Previously, we identified the person or group responsible for validating the model. Now we need to show them the model and make sure it is correct. Often at this stage after reviewing the model, we go back to Step 1 and make some changes and then show the model again. This iterative cycle continues until the model is agreed upon by the validator and deemed correct so that we can move on to MongoDB implementation.

STEP 5: IMPLEMENT

We're done with our modeling, and now we can start creating documents within each of these collections!

Key Points

- Physical data modeling is the process of capturing the detailed technical solution. This is the first time we are actually concerning ourselves with technology and technology issues such as performance, storage, and security.
- There are five steps to physical data modeling: embed or reference (Step 1), accommodate history (Step 2), index and shard (Step 3), review and confirm (Step 4), and implement (Step 5).
- Follow the five guidelines to determine whether to embed or reference.
- The primary and alternate keys from our logical data model are converted into unique indexes on the physical. We can also add additional indexes for performance, called secondary or non-unique indexes.

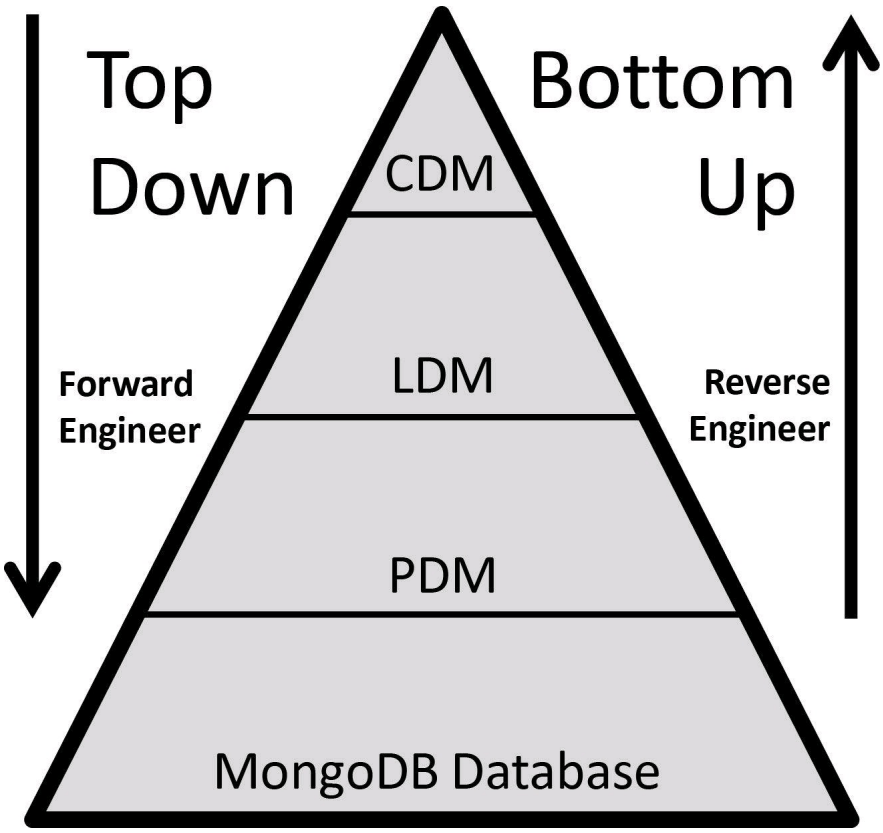
- A star schema is when each set of tables that make up a dimension is flattened into a single table.

- Partitioning (called “sharding” in MongoDB) is when a collection is split up into two or more parts. Vertical partitioning is when fields are split up, and horizontal is when documents are split up. Frequently, both horizontal and vertical are used together.

Section III

Case Study

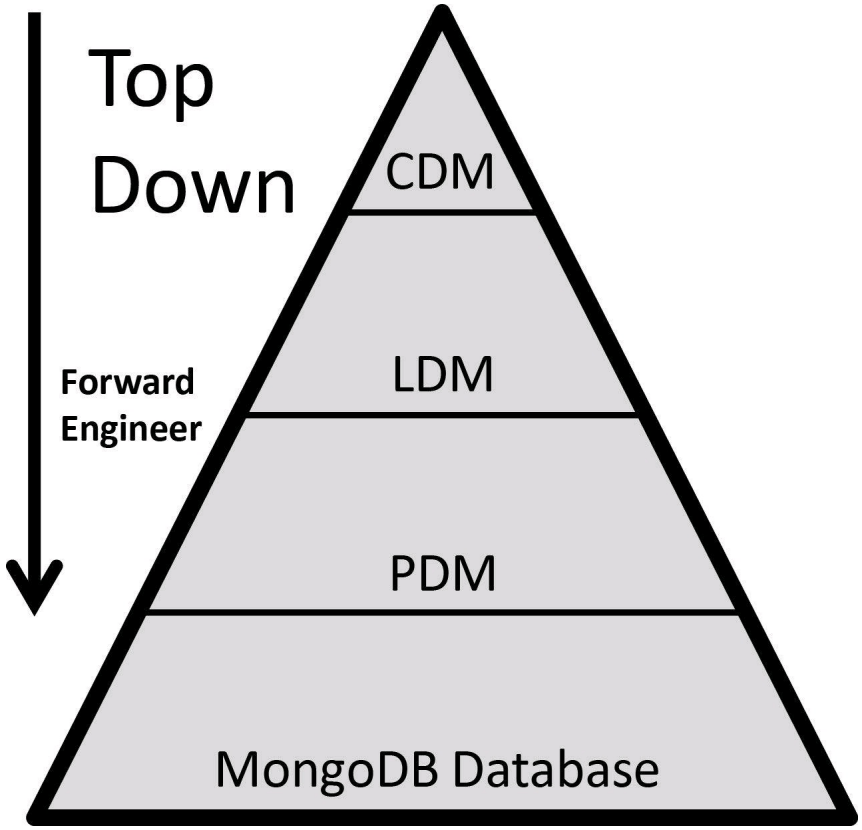
In this section we will walk through a case study to tie together the conceptual, logical, and physical techniques from Section II. We can either choose a "top down" or "bottom up" approach when building a MongoDB database:



The “top down” approach (frequently called “forward engineering”) means starting from the conceptual data model and working our way towards the final physical solution—in this case, a MongoDB database. The “bottom up” approach (frequently called “reverse engineering”) means starting with a MongoDB database and working our way up towards the conceptual data model. We take the bottom-up approach when we did not initially do modeling in building the application, or the data models built were not maintained. After the application has been built, it will need to be supported, and the data models are an essential tool for the business and support teams to understand the structures in the application. In the following case study, we will take the top-down approach, which is the most common approach taken.

Chapter 8
Survey Data Entry

In this case study, we are going to design the MongoDB database for an application that stores the results of surveys using the top-down approach. The top-down approach, also known as forward engineering, means that we are starting with the conceptual data model and ending with a MongoDB database:

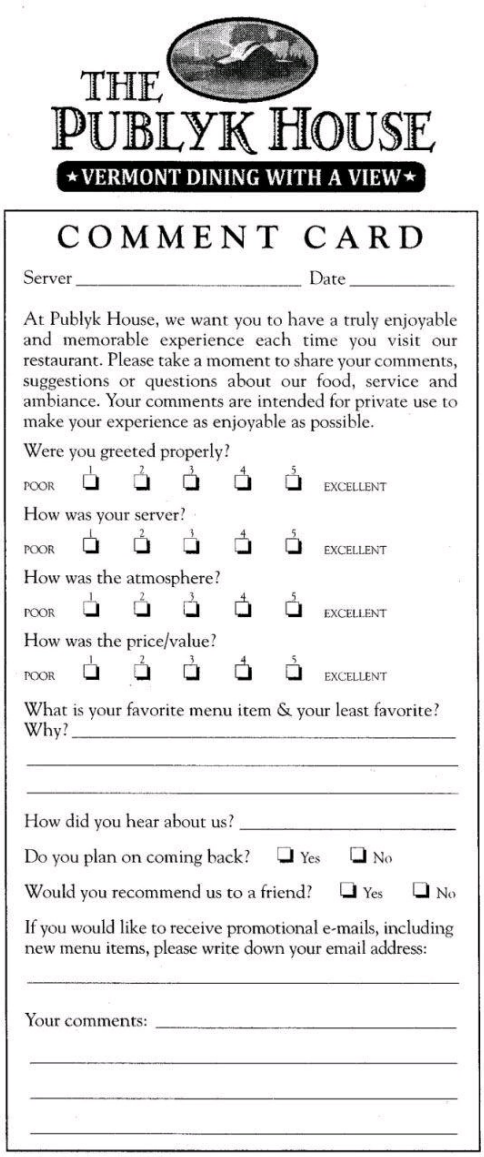


We will first build the conceptual data model (CDM) to capture the scope of the application through key concepts and relationships. Then, using the CDM as our starting point, we will complete the logical data model (LDM) to capture the detailed

business solution. Next, using the LDM as our starting point, we will complete the physical data model (PDM) to capture the detailed technical solution, leveraging embedding, referencing, sharding, and indexing. The PDM can then be implemented as collections in a MongoDB database.

Case Study Background

You are on a project team responsible for building an application that stores the results of surveys. Any organization with any type of survey need (e.g., consumer satisfaction, market research, employee productivity) across any industry (e.g., entertainment, travel, or healthcare) is a potential user of this application. Here is a sample consumer satisfaction survey form for an organization (a restaurant in Vermont) that hired us to store their survey results:



THE PUBLYK HOUSE

★VERMONT DINING WITH A VIEW★

COMMENT CARD

Server _____ Date _____

At Publyk House, we want you to have a truly enjoyable and memorable experience each time you visit our restaurant. Please take a moment to share your comments, suggestions or questions about our food, service and ambiance. Your comments are intended for private use to make your experience as enjoyable as possible.

Were you greeted properly?

POOR ❑1 ❑2 ❑3 ❑4 ❑5 EXCELLENT

How was your server?

POOR ❑1 ❑2 ❑3 ❑4 ❑5 EXCELLENT

How was the atmosphere?

POOR ❑1 ❑2 ❑3 ❑4 ❑5 EXCELLENT

How was the price/value?

POOR ❑1 ❑2 ❑3 ❑4 ❑5 EXCELLENT

What is your favorite menu item & your least favorite?
Why? _____

_____

_____

How did you hear about us? _____

Do you plan on coming back? ❑ Yes ❑ No

Would you recommend us to a friend? ❑ Yes ❑ No

If you would like to receive promotional e-mails, including new menu items, please write down your email address:

_____

Your comments: _____

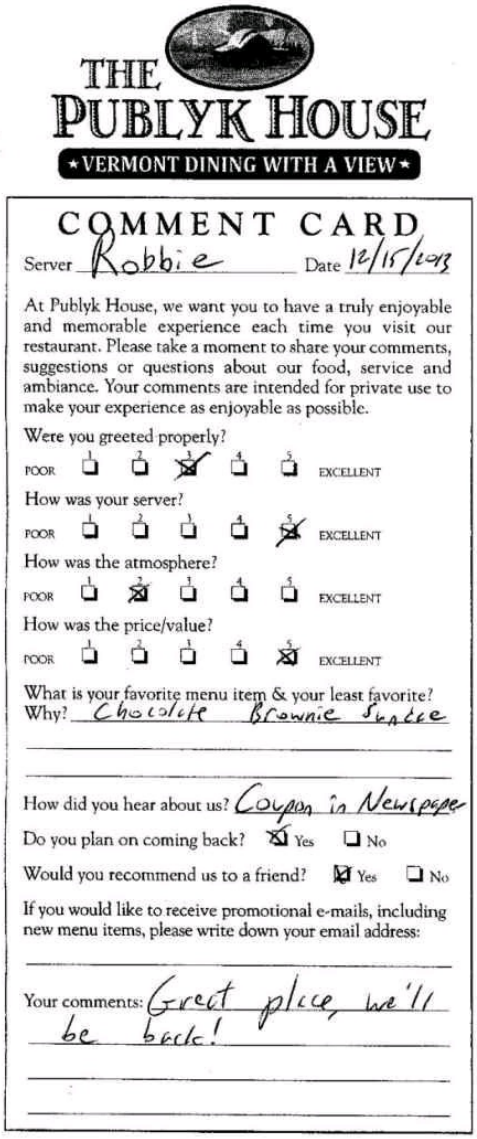
_____

_____

_____

Using this sample survey as a guide, we will start with the conceptual data model and

end with a MongoDB design. In addition to this sample survey template, we are provided with two sample completed surveys:



THE PUBLYK HOUSE

★VERMONT DINING WITH A VIEW★

COMMENT CARD

Server Robbie Date 12/15/2013

At Publyk House, we want you to have a truly enjoyable and memorable experience each time you visit our restaurant. Please take a moment to share your comments, suggestions or questions about our food, service and ambiance. Your comments are intended for private use to make your experience as enjoyable as possible.

Were you greeted properly? POOR 1 2 3 4 5 EXCELLENT

How was your server? POOR 1 2 3 4 5 EXCELLENT

How was the atmosphere? POOR 1 2 3 4 5 EXCELLENT

How was the price/value? POOR 1 2 3 4 5 EXCELLENT

What is your favorite menu item & your least favorite?
Why? Chocolate Brownie Sundae

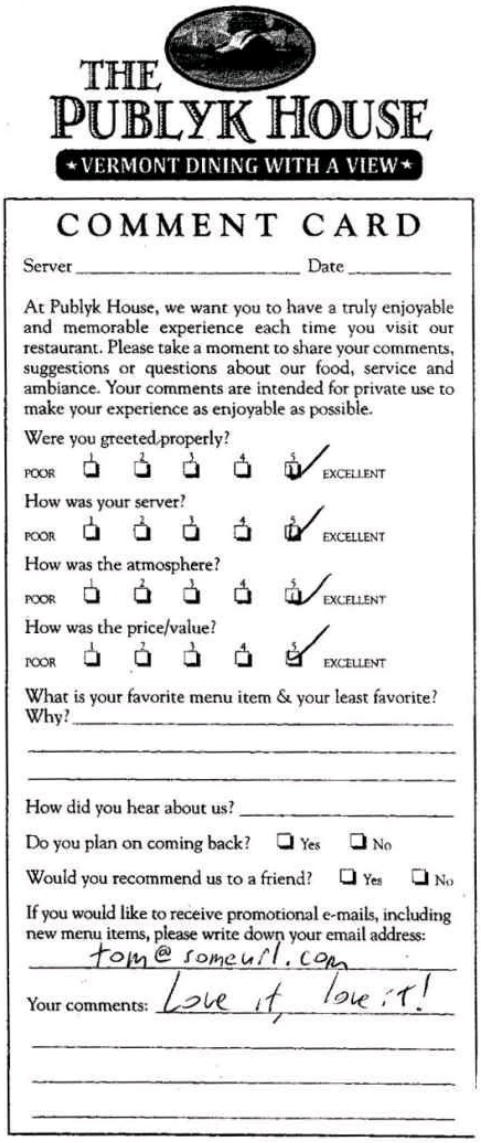
How did you hear about us? Coupon in Newspaper

Do you plan on coming back? ☒ Yes ☐ No

Would you recommend us to a friend? ☒ Yes ☐ No

If you would like to receive promotional e-mails, including new menu items, please write down your email address:

Your comments: Great place, we'll be back!



THE PUBLYK HOUSE

★VERMONT DINING WITH A VIEW★

COMMENT CARD

Server _____ Date _____

At Publyk House, we want you to have a truly enjoyable and memorable experience each time you visit our restaurant. Please take a moment to share your comments, suggestions or questions about our food, service and ambiance. Your comments are intended for private use to make your experience as enjoyable as possible.

Were you greeted properly? POOR 1 2 3 4 5 EXCELLENT

How was your server? POOR 1 2 3 4 5 EXCELLENT

How was the atmosphere? POOR 1 2 3 4 5 EXCELLENT

How was the price/value? POOR 1 2 3 4 5 EXCELLENT

What is your favorite menu item & your least favorite?
Why? _____

How did you hear about us? _____

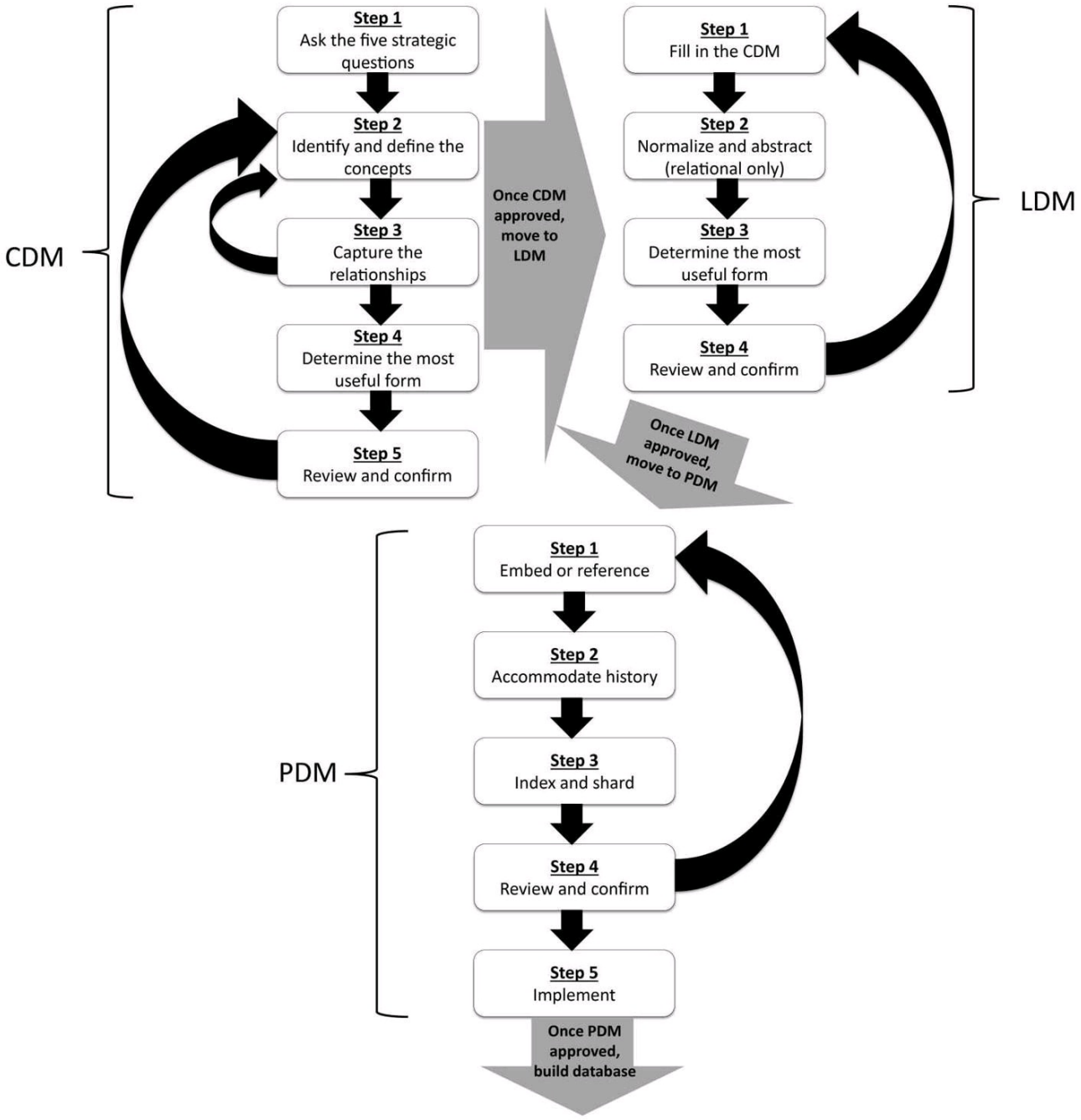
Do you plan on coming back? ☐ Yes ☐ No

Would you recommend us to a friend? ☐ Yes ☐ No

If you would like to receive promotional e-mails, including new menu items, please write down your email address:
tom@someurl.com

Your comments: Love it, love it!

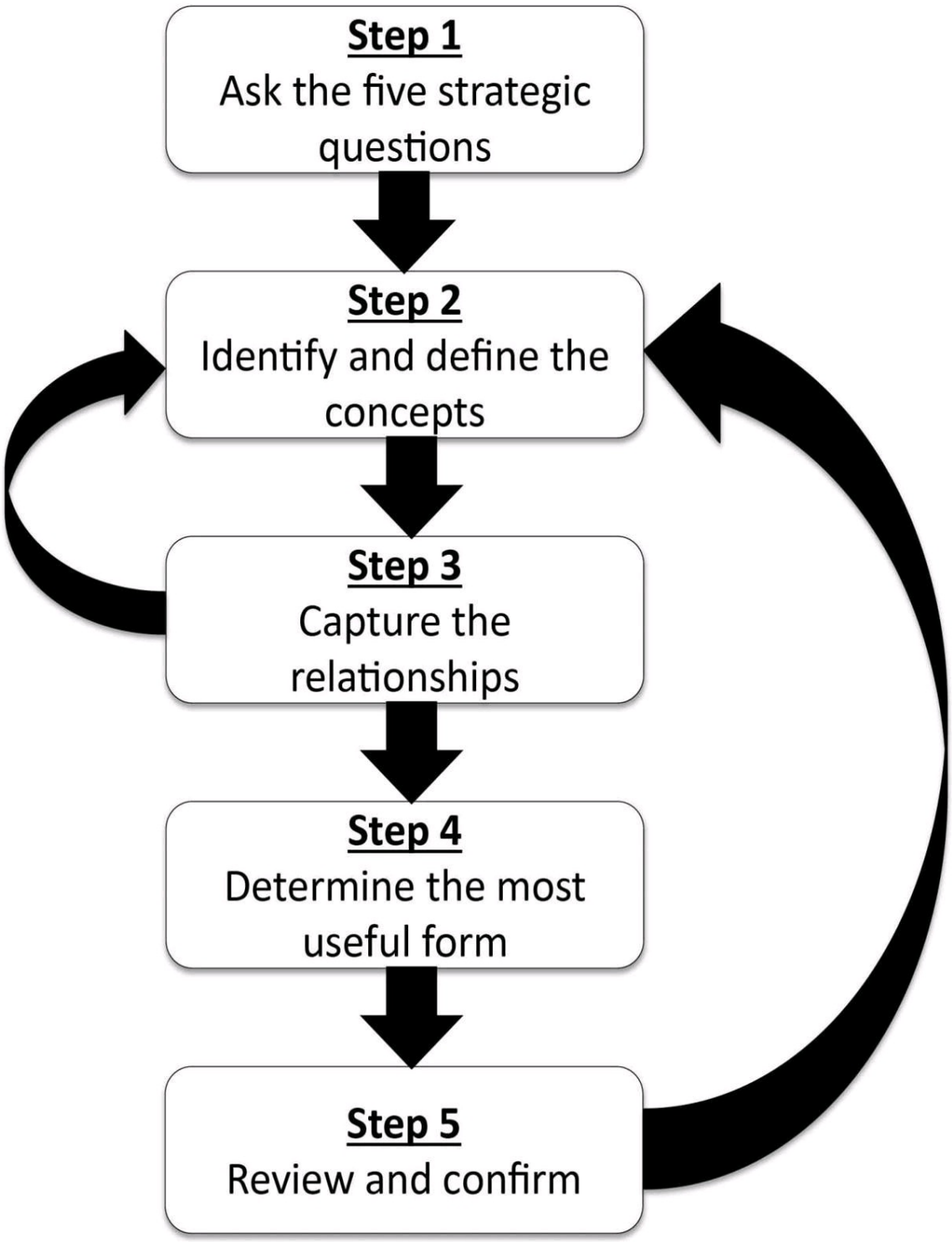
Recall the steps to build the conceptual data model from Chapter 5, the logical data model from Chapter 6, and the physical data model from Chapter 7:



We will complete each of these 14 steps in this case study.

Conceptual Data Modeling

Recall the five steps during the conceptual data modeling phase from Chapter 5:



STEP 1: ASK THE FIVE STRATEGIC QUESTIONS

We met with the project sponsor and her team and asked the five strategic questions. Here are the responses we received:

1. What is the application going to do?

This application is going to allow any organization in any industry to store the results of any type of survey and provide the foundation for us to analyze the results (analysis will be the focus of the next project phase). We think this would be a useful application to many organizations whose surveys are still heavily paper intensive and therefore difficult to store and eventually analyze.

2. "As is" or "to be"?

The application we are building is a brand-new system, and therefore we need a "to be" solution.

3. Is analytics a requirement?

No, this is a data entry application and there is no need for analytics or reporting in this application, which will take place in the next project phase.

4. Who is the audience?

The business analysts group needs to validate our results and the data modeling group will take our model forward to the logical data modeling phase, so the data modeling group represents our users.

5. Flexibility or simplicity?

Flexibility is more important than usability. Over time survey questions will be rephrased or removed and new questions will be added. Any organization should be able to come up with any type of survey with any type of survey question and our system should be able to gracefully accommodate.

STEP 2: IDENTIFY AND DEFINE THE CONCEPTS

From Question 3, we learn that we are building a data entry application, which means our data model will be relational and not dimensional. Therefore we can complete the Concept template:

Survey Concepts

Who?	What?	When?	Where?	Why?	How?
1. Organization 2. Industry 3. Survey Respondent	1. Survey 2. Survey Category 3. Survey Section 4. Survey Question	1. Survey Completion Date			1. Completed Survey

Here are the definitions for each of these concepts:

Completed Survey One filled in survey that contains a collection of opinions from a survey respondent in reaction to an organization's product or service.

Industry The general sector in which an organization operates. This can include retail, manufacturing, healthcare, public welfare, education, etc. It is common for an organization to operate in more than one industry.

Organization The company or government agency that needs the survey. Organization is equivalent to the concept of customer as the organization pays us for the survey management service we provide. We prefer the term "organization" because of the ambiguity often associated with the term "customer."

Survey A template of questions designed to be completed by an individual for continuous improvement of the organization. A completed survey is one filled in survey.

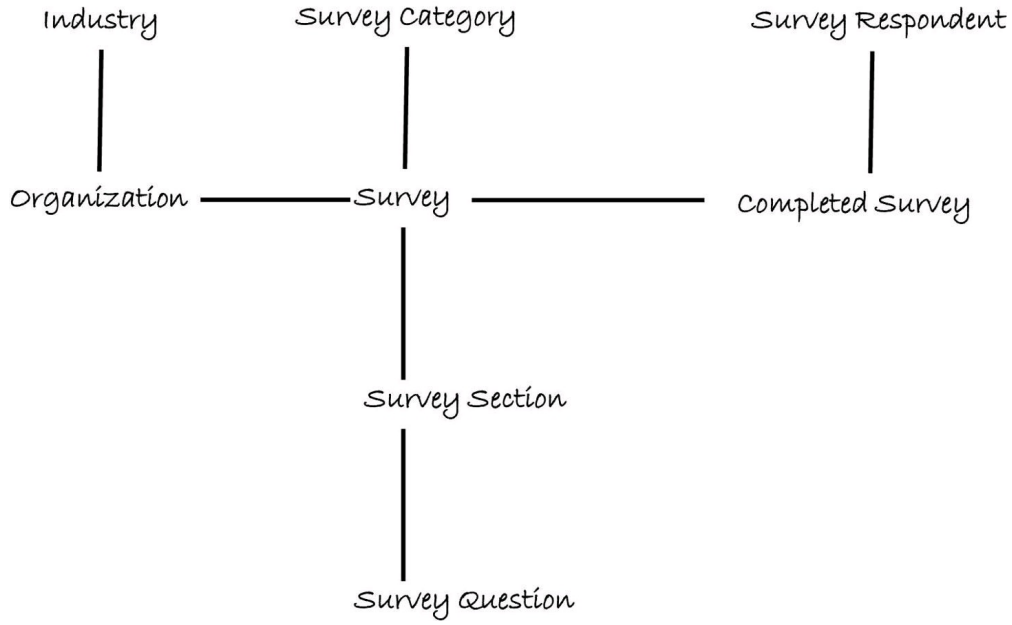
Survey Category The driver for the survey such as employee satisfaction, consumer feedback, course evaluation, market research, etc.

Survey The date that an individual filled in the survey. If there is no date on the survey, this date field

Completion Date	defaults to the date the survey form is received by the application.
Survey Question	The inquiry an organization uses to seek feedback. Questions can be either rating questions, fixed response non-rating questions such as Gender, or free-form text questions.
Survey Respondent	The individual who completes the survey form. We may know little (if anything) about this person other than that he or she completed a survey form. This person does not have to be a customer of the organization to complete a survey.
Survey Section	A logical grouping within the survey. For a hotel, for example, sections can include room service, dining, check-in experience, etc.

STEP 3: CAPTURE THE RELATIONSHIPS

After meeting with the business user, we created the following sketch on a flipchart:



Notice that the concept of **Survey Completion Date** was too detailed for this conceptual data model, so this date will instead become an attribute on our logical data model.

Once we've identified the fact that two entities have a relationship, we need to articulate what that relationship looks like and make it precise. To make it precise, we find ourselves asking up to eight questions: two on participation, two on optionality, and up to four questions on subtyping. So after creating a template for our Survey

example, I asked a business expert each question and checked off the appropriate answer:

Can an Industry contain more than one Organization?	✓	
Can an Organization operate in more than one Industry?	✓	
Can an Industry exist without an Organization?	✓	
Can an Organization exist without an Industry?		✓
Are there examples of Industry that would be valuable to show?		✓
Are there examples of Organization that would be valuable to show?		✓
Does an Industry go through a lifecycle?		✓
Does a Customer go through a lifecycle?		✓
Can an Organization create more than one Survey?	✓	
Can a Survey be created by more than one Organization?	✓	

Can an Organization exist without a Survey?	✓	
Can a Survey exist without an Organization?		✓
Are there examples of Survey that would be valuable to show?		✓
Does a Survey go through a lifecycle?	✓	
Can a Survey Category contain more than one Survey?	✓	
Can a Survey belong to more than one Survey Category?		✓
Can a Survey Category exist without a Survey?	✓	
Can a Survey exist without a Survey Category?		✓
Are there examples of Survey Category that would be valuable to show?		✓
Does a Survey Category go through a lifecycle?		✓
Can a Survey contain more than one Survey Section?	✓	
Can a Survey Section belong to more than one Survey?		✓
Can a Survey exist without a Survey Section?	✓	

Can a Survey Section exist without a Survey?		✓
Are there examples of Survey Section that would be valuable to show?		✓
Does a Survey Section go through a lifecycle?		✓
Can a Survey Section contain more than one Survey Question?	✓	
Can a Survey Question belong to more than one Survey Section?		✓
Can a Survey Section exist without a Survey Question?	✓	
Can a Survey Question exist without a Survey Section?		✓
Are there examples of Survey Question that would be valuable to show?		✓
Does a Survey Question go through a lifecycle?		✓
Can a Survey be the template for more than one Completed Survey?	✓	
Can a Completed Survey use more than one Survey?		✓
Can a Survey exist without a Completed Survey?	✓	

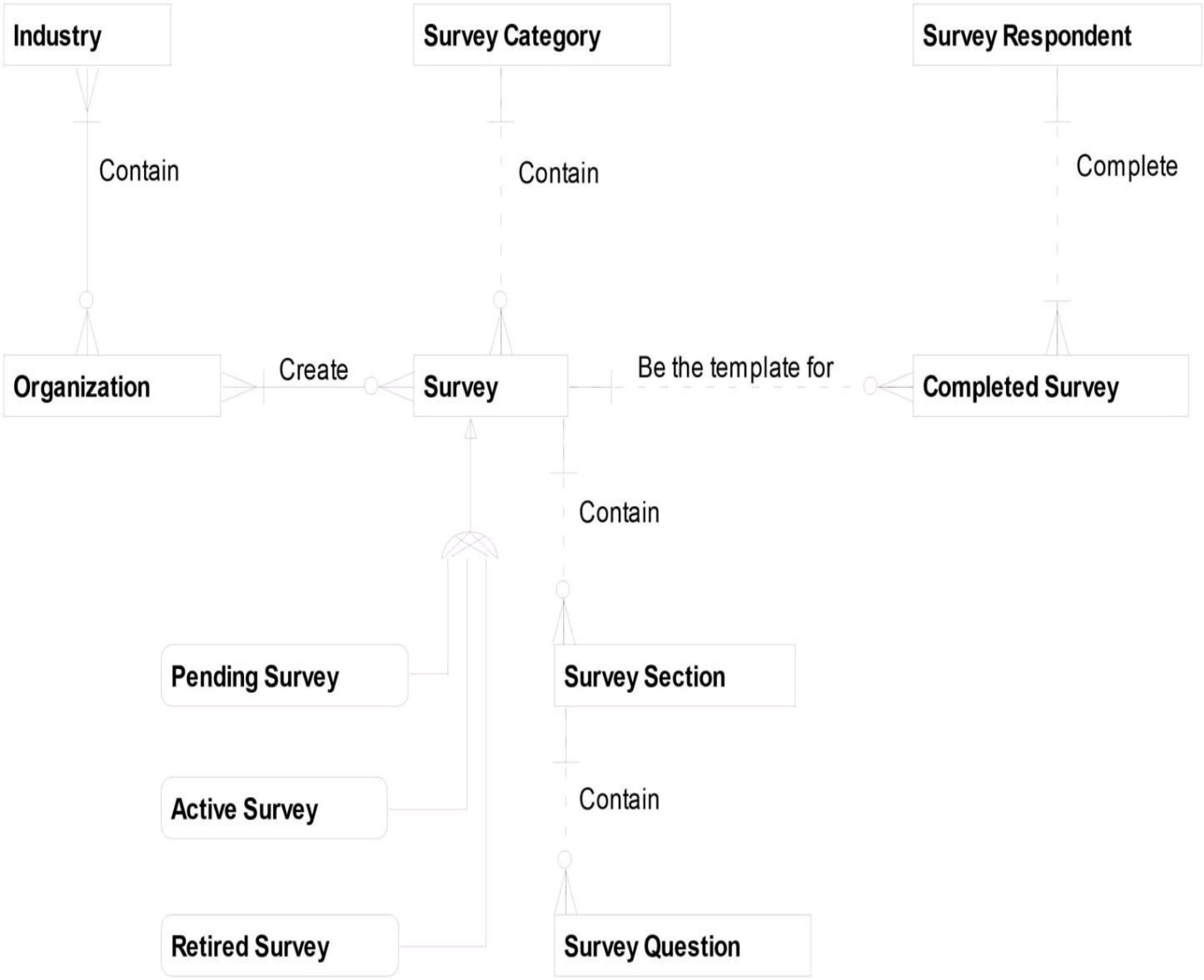
Can a Completed Survey exist without a Survey?		✓
Are there examples of Completed Survey that would be valuable to show?		✓
Does a Completed Survey go through a lifecycle?		✓
Can a Survey Respondent complete more than one Completed Survey?	✓	
Can a Completed Survey be completed by more than one Survey Respondent?		✓
Can a Survey Respondent exist without a Completed Survey?		✓
Can a Completed Survey exist without a Survey Respondent?		✓
Are there examples of Survey Respondent that would be valuable to show?		✓
Does a Survey Respondent go through a lifecycle?		✓

Now that these questions are answered, we can create the most useful form.

STEP 4: DETERMINE THE MOST USEFUL FORM

Recall the answer to Strategic Question #4 from the list of five strategic questions from Step 1: *Who is our audience?* We know that the business analyst group are our validators and the data modeling group will be our main users for this model. After having a conversation with both groups, we learn that using the traditional data modeling notation would be the most valuable form for communication.

We can therefore translate the answers to the previous set of questions into the following relational conceptual data model:



Note that from these questions, we learned that **Survey** goes through a lifecycle. Therefore, after a discussion with the business, we added three subtypes to the model to capture the **Survey** lifecycle: **Pending Survey**, **Active Survey**, and **Retired Survey**. We would also need to define each of these entities.

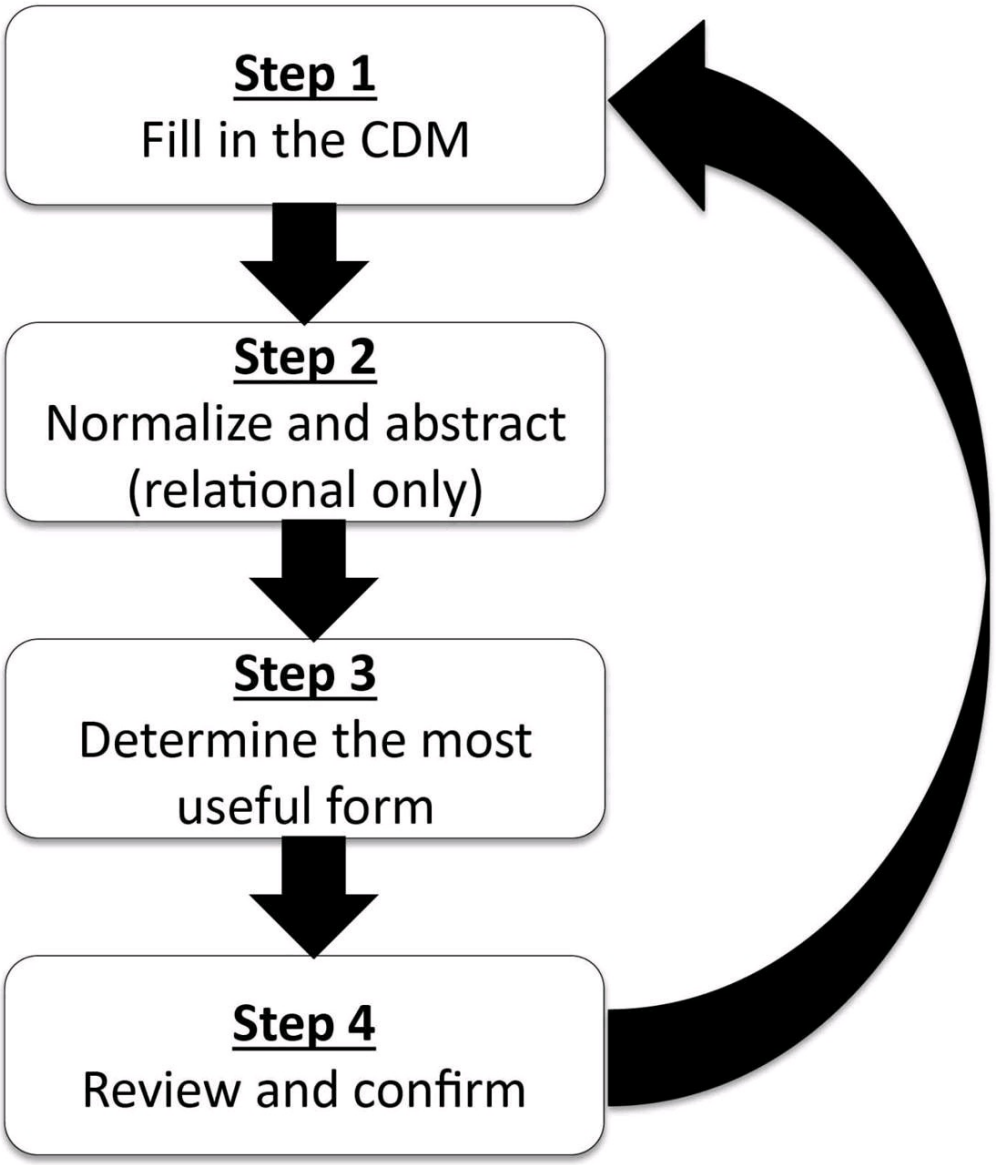
Spend a few minutes going through the answers to the questions from the prior step and see how they translate into this model.

STEP 5: REVIEW AND CONFIRM

We now review this model with the business analyst group. This may require us making changes to our model and iterating through these steps before moving on to the next phase of logical data modeling.

Logical Data Modeling

Recall the five steps during the logical data modeling phase from Chapter 6:



STEP 1: FILL IN THE CDM

We fill in the CDM by identifying and defining the properties of each concept, which involves completing the Properties Template and Property Characteristics Template. The Properties Template captures the properties (also known as attributes or data elements) of each conceptual entity. Here is the completed Properties Template for our survey application:

Survey Properties

	Industry	Organization	Survey		Survey Question	Completed Survey
			Survey Category	Survey Section		Survey Respondent
Name	Industry Name	Organization Name	Survey Category Name	Survey Section Name	Survey Question Label Name	Survey Respondent Name
Text				Survey Section Description	Survey Question Description, Possible Answer value, Possible Answer Description	Completed Survey Free Form Response Text, Completed Survey Fixed Response Answer
Amount						
Date		Organization First Survey Date				Completed Survey Date
Code	SIC Code		Survey Code	Survey Category Code		
Quantity						
Number		Organization DUNS Number			Survey Question Number	
Identifier						Survey Respondent ID
Indicator					Survey Question Singular Response Indicator	
Rate						

Percent

Complex

The second template, the Property Characteristics Template, captures important metadata on each property. Here is the completed Property Characteristics Template for our survey application (Note that I removed the Source and Transform columns to make this spreadsheet more readable):

Survey Property Characteristics

Property	Definition	Sample Values	Format	Length
Industry Name	The common term used to describe the general sector an organization operations within. This is the standard description for the SIC code.	Dairy Farms Copper Ores Dog and Cat Food	Char	50
SIC Code	The Standard Industry Classification (SIC) is a system for classifying industries by a four-digit code. Established in the United States in 1937, it is used by government agencies to classify industry areas.	0241 [for Dairy Farms] 1021 [for Copper Ores] 2047 [for Dog and Cat Food]	Char	4
Organization Name	The common term used to describe the company or government agency that needs the survey.	IBM Walmart Paul's Diner Publyk House	Char	50
Organization First Survey Date	The date when the organization first started using our services. This is not the date we received payment, but the date the organization launched their first survey.	Apr-25-2014	Date	

Organization DUNS Number	Dun & Bradstreet (D&B) provides a DUNS Number, a unique nine digit identification number, for each organization.	123456789	Char	9
Survey Code	The unique and required short term the business uses for referring to a survey.	M329-A	Char	6
Survey Category Name	The common term used to describe the driver for the survey.	Employee Satisfaction Consumer Feedback	Char	50
Survey Category Code	The unique and required short term to describe the driver for the survey such as employee satisfaction.	ES CF	Char	2
Survey Section Name	The common term used to describe the logical grouping within the survey.	General Hotel Room Dining Experience	Char	50
Survey Section Description	The detailed text explaining the logical grouping within the survey. This is not displayed on the survey form.	This section contains those questions pertaining to the quality of the food and overall dining experience.	Char	255
Survey Question Label Name	What the survey respondent sees on the form. That is, the question that appears.	How was the atmosphere? This question lets the	Char	255

Survey Question Description	An explanation or background on the question, which is not displayed on the survey form.	respondent rate the quality of the decorating and warmness of the room.	Char	255
Possible Answer Value	Certain questions have a fixed response such as the Gender question for “Male” or “Female” or the “From 1 to 5, how was your meal?” question. This field stores all of the possible fixed responses.	Male Female 3	Char	50
Possible Answer Description	Certain questions have a fixed response such as the Gender question for “Male” or “Female” or the “From 1 to 5, how was your meal?” question. This field stores the meaning for each of the Possible Answer Values. This field is not displayed on the survey form.	1 means Poor 3 means Average	Char	100
Survey Question Number	A required number assigned to each question. This number is unique within a survey.	1 2	Num	3
Survey Question Singular Response Indicator	Some questions allow for more than one response. This indicator when set to “Y” for “Yes” captures that only one response is expected for this particular question. This indicator when set to “N” for “No” captures that more than one response can be chosen for this particular question.	Y N	Boolean	
Survey Respondent Name	The name the survey respondent writes on the completed survey. This name is not validated and can just contain the first name, just the last name, or both.	Bob Jones Jones Bob	Char	100
Survey Respondent ID	A unique and required system-generated value for each survey respondent.	123456	Num	6

Completed Survey Free Form Response Text	Captures the responses to those questions that do not require a fixed response such as the answer to "What was your favorite dessert?"	Brownie Sundae	Char	255
Completed Survey Fixed Response Answer	Captures the responses to those questions that require a fixed response such as the gender question.	Male Female	Char	100
Completed Survey Date	The date the survey respondent completed the survey. If this date is unknown, this will be the date the survey data is received by the application.	Apr-15-2014	Date	

STEP 2: NORMALIZE AND ABSTRACT (RELATIONAL ONLY)

The model we are building is relational, and therefore we will need to normalize and also decide where (if at all) to apply abstraction.

First Normal Form (1NF)

For first normal form, we need to move repeating attributes to a new entity and also separate multi-valued attributes. Following the question templates from Chapter 6, we asked the business analysts these questions and received answers for each:

Can an **Industry** have more than one **Industry Name**?	✓
Does an **Industry Name** contain more than one piece of business information?	✓
Can an **Industry** have more than one **SIC Code**?	✓

Does an **SIC Code** contain more than one piece of business information?	✓
Can an **Organization** have more than one **Organization Name**?	✓
Does an **Organization Name** contain more than one piece of business information?	✓
Can an **Organization** have more than one **Organization First Survey Date**?	✓
Does an **Organization First Survey Date** contain more than one piece of business information?	✓
Can an **Organization** have more than one **Organization DUNS Number**?	✓
Does an **Organization DUNS Number** contain more than one piece of business information?	✓
Can a **Survey** have more than one **Survey Code**?	✓
Does a **Survey Code** contain more than one piece of business information?	✓
Can a **Survey Category** have more than one **Survey Category Name**?	✓
Does a **Survey Category Name** contain more than one piece of business information?	✓
Can a **Survey** have more than one **Survey Category Code**?	✓
Does a **Survey Category Code** contain more than one piece of business information?	✓

Can a **Survey Section** have more than one **Survey Section Name**?	✓
Does a **Survey Section Name** contain more than one piece of business information?	✓
Can a **Survey Section** have more than one **Survey Section Description**?	✓
Does a **Survey Section Description** contain more than one piece of business information?	✓
Can a **Survey Question** have more than one **Survey Question Label Name**?	✓
Does a **Survey Question Label Name** contain more than one piece of business information?	✓
Can a **Survey Question** have more than one **Survey Question Description**?	✓
Does a **Survey Question Description** contain more than one piece of business information?	✓
Can a **Survey Question** have more than one **Possible Answer Value**?	✓
Does a **Possible Answer Value** contain more than one piece of business information?	✓
Can a **Survey Question** have more than one **Possible Answer Description**?	✓
Does a **Possible Answer Description** contain more than one piece of business information?	✓

Can a **Survey Question** have more than one **Survey Question Number**? ✓

Does a **Survey Question Number** contain more than one piece of business information? ✓

Can a **Survey Question** have more than one **Survey Question Singular Response Indicator**? ✓

Does a **Survey Question Singular Response Indicator** contain more than one piece of business information? ✓

Can a **Survey Respondent** have more than one **Survey Respondent Name**? ✓

Does a **Survey Respondent Name** contain more than one piece of business information? ✓

Can a **Survey Respondent** have more than one **Survey Respondent ID**? ✓

Does a **Survey Respondent ID** contain more than one piece of business information? ✓

Can a **Completed Survey** have more than one **Completed Survey Free Form Response Text**? ✓

Does a **Completed Survey Free Form Response Text** contain more than one piece of business information? ✓

Can a **Completed Survey** have more than one **Completed Survey Fixed Response Answer**? ✓

Does a **Completed Survey Fixed Response Answer** contain more than one piece of business

information?	✓
Can a **Completed Survey** have more than one **Completed Survey Date**?	✓
Does a **Completed Survey Date** contain more than one piece of business information?	✓

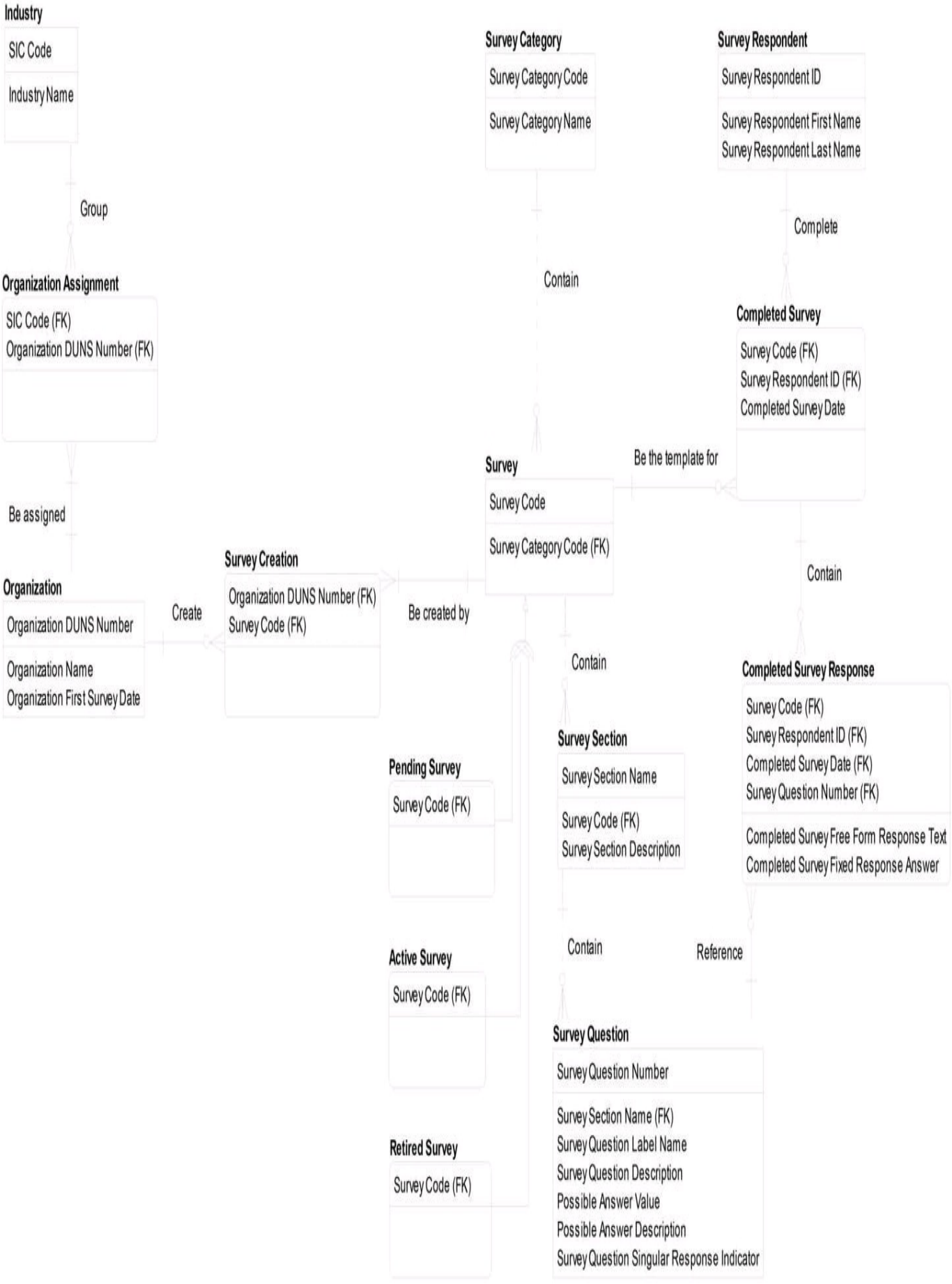
We received an answer of "Yes" to six of these questions:

- An **Organization** can have more than one **Organization Name**. We learn, for example, that a hotel chain such as *Hilton* can sometimes go by the names *Hilton* or *DoubleTree* or *Hampton Inn*. We defined **Organization** as the "company or government agency that needs the survey." So we would need to have the discussion that if *Hampton Inn* would like a survey, would *Hampton Inn* be the **Organization Name** or would the parent company *Hilton* be the **Organization Name**? What's very interesting is that as we start asking these normalization questions we frequently refine our existing terminology and definitions. The business may decide, for example, to store an organization hierarchy based on parent company and subsidiaries. Or the business may decide just to keep track of the organization that is paying for the survey, regardless of whether they a part of a larger organization. Let's assume the business decides to just store the organization that is paying for the survey and not the whole organization hierarchy.
- Both **Organization First Survey Date** and **Completed Survey Date** contain more than one piece of business information. Dates contain a day, month, and year. Year contains a year, decade, century, and millennium. Sometimes we identify attributes that do contain more than one piece of information, but there may be no value in showing the individual pieces. In these cases, it is best to keep the attribute in its original state as with these dates.
- A **Survey Respondent Name** can contain more than one piece of business information. We might, for example, have *Bob Jones* as a name which contains both the respondent's first and last name. Weigh the *value* gained in having the attribute broken down into its components with the *effort* potentially needed to break the attribute down. Sometimes the value may not be worth the effort. The business might determine some value in seeing the components, but if the task is large and highly error prone, it may not be worth the development effort. In this

case, let's assume our conversation with the business ended with breaking down this name into a first and last name.

- A **Completed Survey** can have more than one **Completed Survey Free Form Response Text** and more than one **Completed Survey Fixed Response Answer**. Similar to the survey containing more than one question, the completed survey can also return more than one answered question. For example, if a survey contains five free form questions and ten fixed response questions, it is very possible for the completed survey to also contain fifteen answered questions. If the survey respondent left five of the questions empty, there still would be ten answered questions. We therefore need to ensure we have a many-to-many relationship between **Completed Survey** and **Survey Question**.

Here is our 1NF model after answering these questions, breaking up **Survey Respondent Name** into first and last name, and adding the many-to-many relationship between **Completed Survey** and **Survey Question**:


Industry
SIC Code
Industry Name
Group
Organization Assignment
SIC Code (FK)
Organization DUNS Number (FK)
Be assigned
Organization
Organization DUNS Number
Organization Name
Organization First Survey Date
Create
Survey Creation
Organization DUNS Number (FK)
Survey Code (FK)
Be created by
Survey Category
Survey Category Code
Survey Category Name
Contain
Survey
Survey Code
Survey Category Code (FK)
Be the template for
Survey Respondent
Survey Respondent ID
Survey Respondent First Name
Survey Respondent Last Name
Complete
Completed Survey
Survey Code (FK)
Survey Respondent ID (FK)
Completed Survey Date
Contain
Completed Survey Response
Survey Code (FK)
Survey Respondent ID (FK)
Completed Survey Date (FK)
Survey Question Number (FK)
Completed Survey Free Form Response Text
Completed Survey Fixed Response Answer
Reference
Contain
Survey Section
Survey Section Name
Survey Code (FK)
Survey Section Description
Contain
Survey Question
Survey Question Number
Survey Section Name (FK)
Survey Question Label Name
Survey Question Description
Possible Answer Value
Possible Answer Description
Survey Question Singular Response Indicator
Pending Survey
Survey Code (FK)
Active Survey
Survey Code (FK)
Retired Survey
Survey Code (FK)

Note that there might be additional attributes in the **Survey** subtypes that need to be identified.

Second Normal Form (2NF)

For second normal form, we need to ensure that each entity contains the minimal set of attributes that uniquely identifies each entity instance. In other words, each entity must contain the minimal primary key. We therefore complete the following template:

Is a **SIC Code** needed to retrieve a single instance of **Industry Name**?	✓	
Is an **Organization DUNS Number** needed to retrieve a single instance of **Organization Name**?	✓	
Is an **Organization DUNS Number** needed to retrieve a single instance of **Organization First Survey Date**?	✓	
Is a **Survey Category Code** needed to retrieve a single instance of **Survey Category Name**?	✓	
Is a **Survey Section Name** needed to retrieve a single instance of **Survey Section Description**?		✓
Is a **Survey Question Number** needed to retrieve a single instance of **Survey Question Label Name**?	✓	
Is a **Survey Question Number** needed to retrieve a single instance of **Survey Question Description**?	✓	

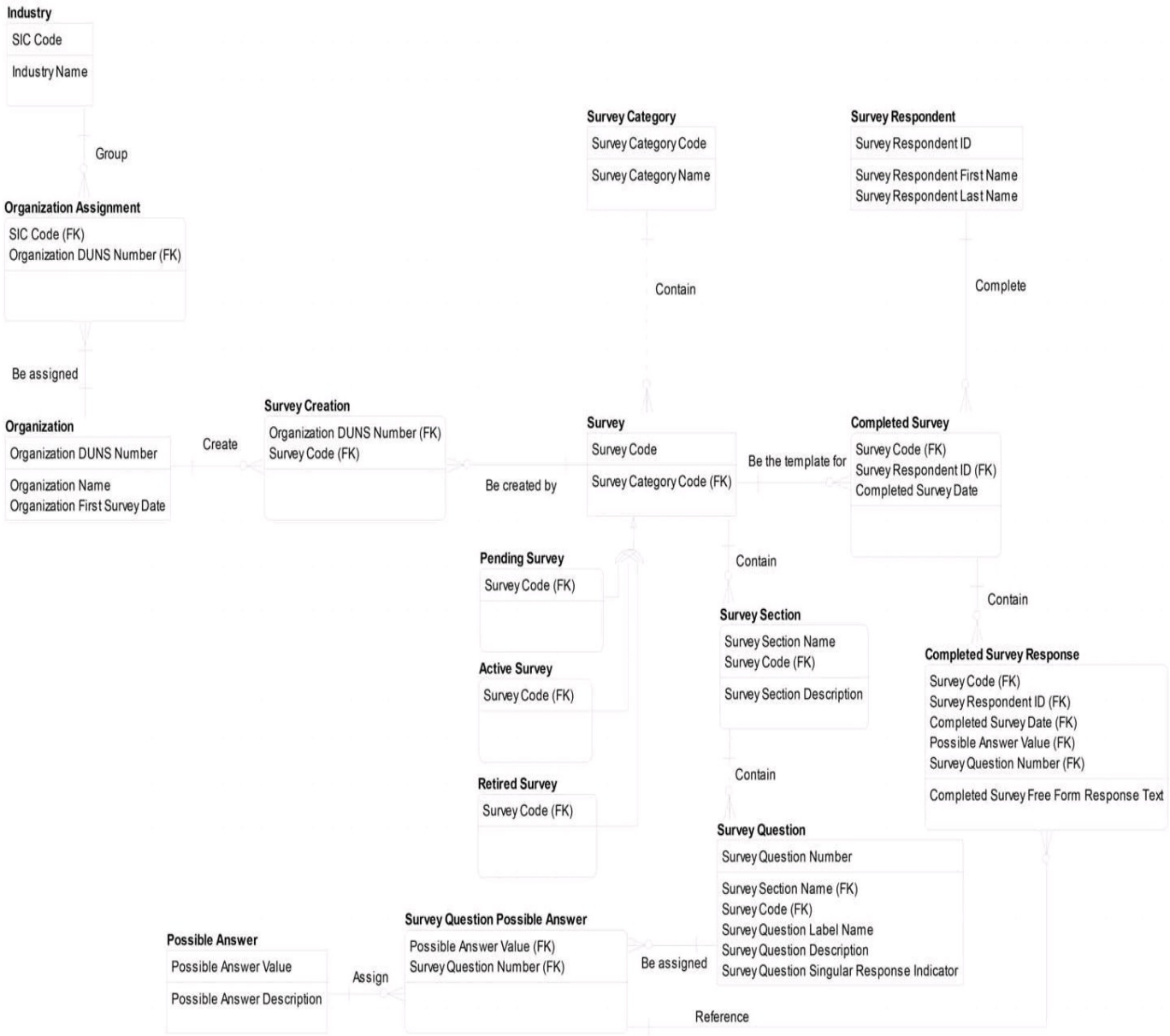
Is a **Survey Question Number** needed to retrieve a single instance of **Possible Answer Value**?	✓
Is a **Survey Question Number** needed to retrieve a single instance of **Possible Answer Description**?	✓
Is a **Survey Question Number** needed to retrieve a single instance of **Survey Question Singular Response Indicator**?	✓
Is a **Survey Respondent ID** needed to retrieve a single instance of **Survey Respondent First Name**?	✓
Is a **Survey Respondent ID** needed to retrieve a single instance of **Survey Respondent Last Name**?	✓
Are a **Survey Code**, **Survey Respondent ID**, **Completed Survey Date**, and **Survey Question Number** needed to retrieve a single instance of **Completed Survey Free Form Response Text**?	✓
Are a **Survey Code**, **Survey Respondent ID**, **Completed Survey Date**, and **Survey Question Number** needed to retrieve a single instance of **Completed Survey Fixed Response Answer**?	✓

For the question on **Survey Section**, we learn that more is needed than just the **Survey Section Name** to retrieve a single instance of **Survey Section Description**. After additional discussion with the business, we learn that the **Survey Code** is needed as well in the **Survey Section** primary key.

We also learn that the **Possible Answer Value** and **Possible Answer Description** need more than just **Survey Question Number** as their primary key. We learn from talking with the business that the same **Possible Answer Value** can apply to more than one **Survey Question**. We also learn that when the **Survey Question Singular Response Indicator** is set to *No*, the same **Survey Question** can contain more than one **Possible Answer**. Therefore there is a many-to-many relationship between

Survey Question and **Possible Answer**, which is resolved on the logical through a linking entity called an associative entity. We can then connect this associative entity to **Completed Survey Response**, which addresses the minimal primary key issue for the last two questions in our spreadsheet where the answer was *No*.

Here is our 2NF model after making the above changes to **Survey Section**, adding the associative entity between **Possible Answer** and **Survey Question**, and connecting **Survey Question Possible Answer** to **Completed Survey Response**:



Third Normal Form (3NF)

For third normal form, we need to remove hidden dependencies. A hidden dependency is when a property of an entity depends upon one or more other properties in that same entity instead of directly on that entity's primary key. We therefore complete the following template for those entities that have more than two attribute:

Is **Organization Name** a fact about any other attribute in **Organization**?	✓
Is **Organization First Survey Date** a fact about any other attribute in **Organization**?	✓
Is **Survey Question Label Name** a fact about any other attribute in **Survey Question**?	✓
Is **Survey Question Description** a fact about any other attribute in **Survey Question**?	✓
Is **Survey Question Singular Response Indicator** a fact about any other attribute in **Survey Question**?	✓
Is **Survey Respondent First Name** a fact about any other attribute in **Possible Answer**?	✓
Is **Survey Respondent Last Name** a fact about any other attribute in **Possible Answer**?	✓

Because there are no hidden dependencies on our data model, there are no changes to make, and the model remains unchanged from our 2NF model.

Abstraction (Relational Only)

Recall that abstraction brings flexibility to a design by redefining and combining some of the attributes, entities, and relationships within the model into more generic terms.

The general question that needs to be asked is:

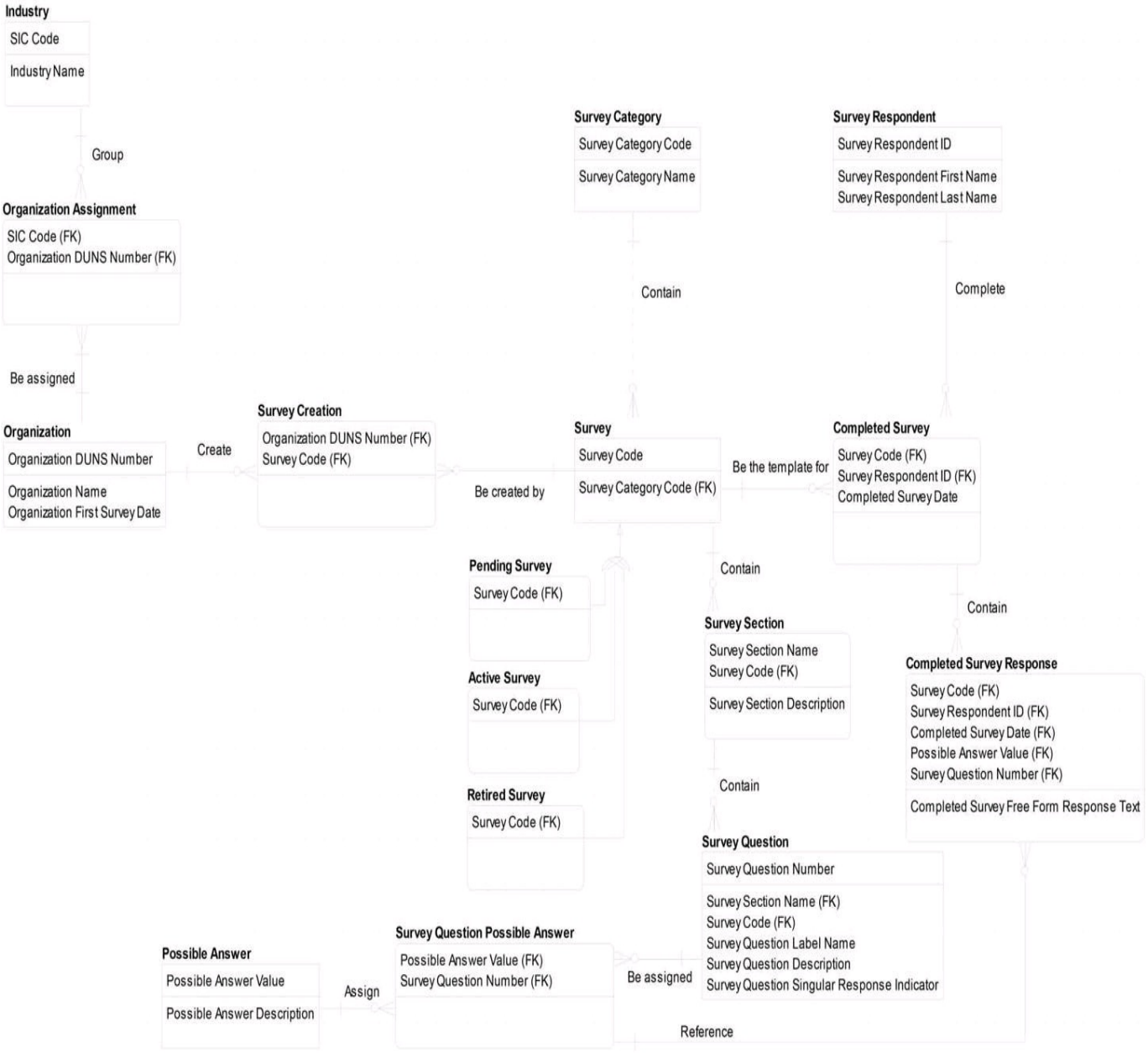
"Are there any entities, relationships, or data elements that should be made more generic to accommodate future requirements?"

For our survey data model, I do not believe there are any valuable opportunities to abstract. We applied some abstraction in viewing everything on the survey form as a

question. We can abstract **Survey Respondent** and **Organization** into the generic **Party** concept, as a **Party** can be either a person or organization, and each **Party** can play many roles such as **Survey Respondent** or **Organization**. But what value would abstracting into **Party** provide us here? We would lose precision and the model would become more difficult to read, and it is difficult to come up with a business scenario where the **Party** concept would be valuable.

STEP 3: DETERMINE THE MOST USEFUL FORM

We determine that the traditional data modeling form would work well for our users, so here is our fully normalized data model, which is the same as our 2NF model:

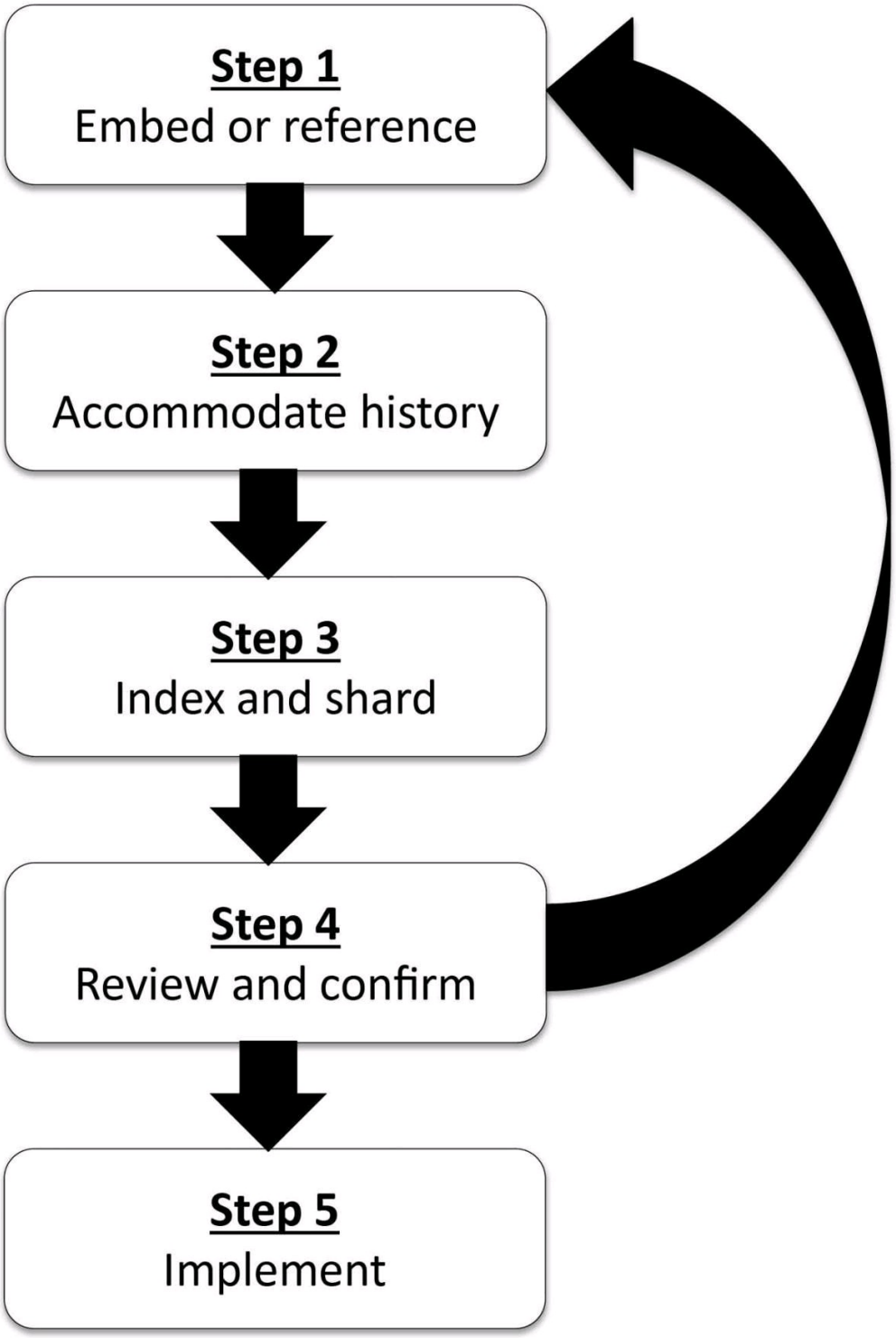


STEP 4: REVIEW AND CONFIRM

Previously, we identified the person or group responsible for validating the model. Now we need to show them the model and make sure it is correct. Often at this stage after reviewing the model, we go back to Step 1 and make some changes and then show the validators the model again. This iterative cycle continues until the model is agreed upon by the validators and deemed correct so that we can move on to the physical modeling phase.

Physical Data Modeling

Recall the five steps during the physical data modeling phase from Chapter 7:



STEP 1: EMBED OR REFERENCE

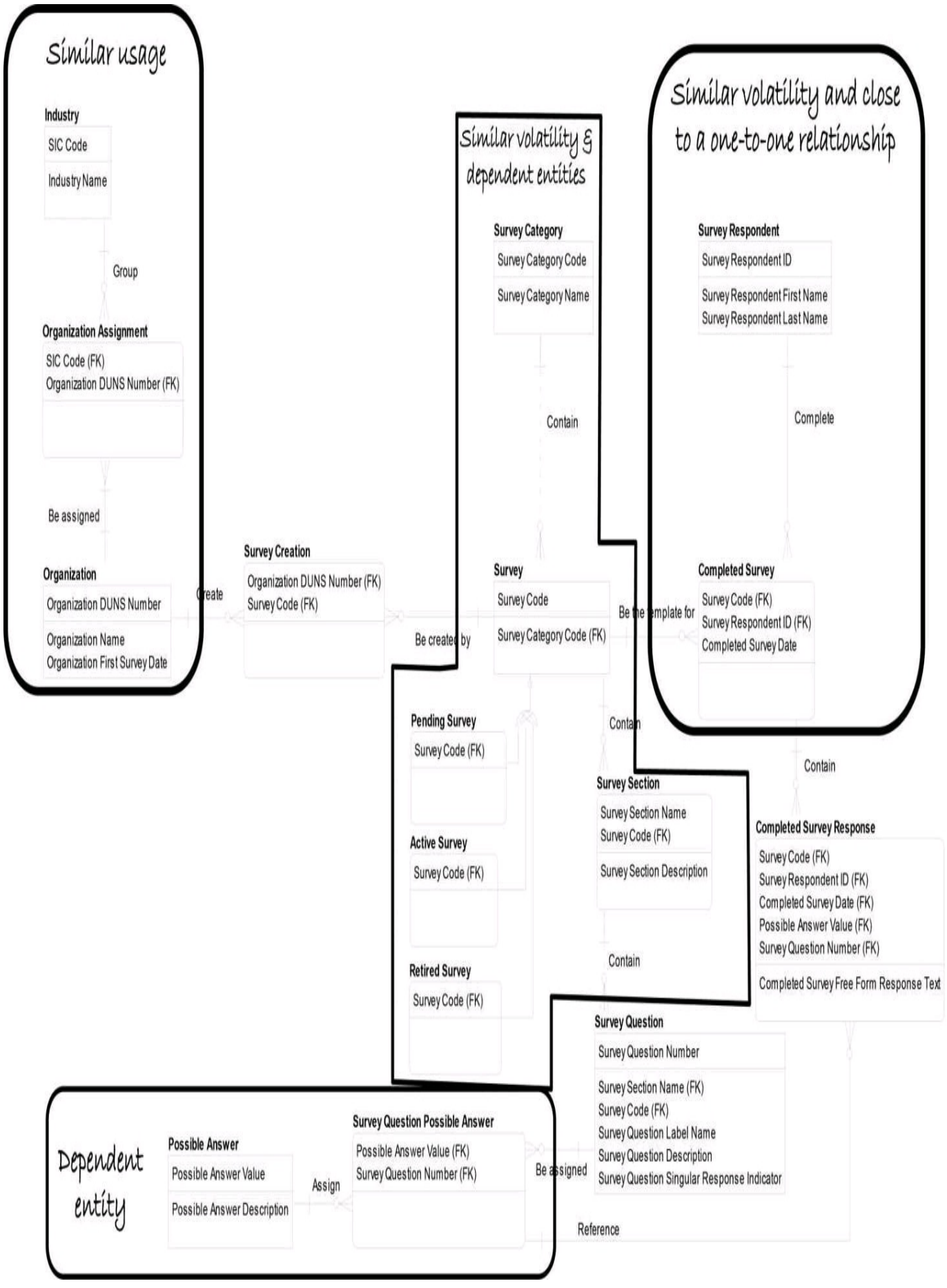
MongoDB resolves relationships in one of two ways: embed or reference. Recall the

top five reasons for embedding over referencing from Chapter 7:

1. Requirements state that data from the related entities are frequently queried together.
2. The child in the relationship is a dependent entity.
3. There is a one-to-one relationship between the two entities.
4. The related entities have similar volatility.

1. The entity you are considering embedding is not a key entity.

With these five reasons in mind, we can take a pencil or marker and group entities together into collections:


Similar usage
Industry
SIC Code
Industry Name
Group
Organization Assignment
SIC Code (FK)
Organization DUNS Number (FK)
Be assigned
Organization
Organization DUNS Number
Organization Name
Organization First Survey Date
Create
Survey Creation
Organization DUNS Number (FK)
Survey Code (FK)
Be created by
Similar volatility & dependent entities
Survey Category
Survey Category Code
Survey Category Name
Contain
Survey
Survey Code
Survey Category Code (FK)
Be the template for
Similar volatility and close to a one-to-one relationship
Survey Respondent
Survey Respondent ID
Survey Respondent First Name
Survey Respondent Last Name
Complete
Completed Survey
Survey Code (FK)
Survey Respondent ID (FK)
Completed Survey Date
Contain
Pending Survey
Survey Code (FK)
Active Survey
Survey Code (FK)
Retired Survey
Survey Code (FK)
Contain
Survey Section
Survey Section Name
Survey Code (FK)
Survey Section Description
Contain
Completed Survey Response
Survey Code (FK)
Survey Respondent ID (FK)
Completed Survey Date (FK)
Possible Answer Value (FK)
Survey Question Number (FK)
Completed Survey Free Form Response Text
Survey Question
Survey Question Number
Survey Section Name (FK)
Survey Code (FK)
Survey Question Label Name
Survey Question Description
Survey Question Singular Response Indicator
Be assigned
Survey Question Possible Answer
Possible Answer Value (FK)
Survey Question Number (FK)
Assign
Possible Answer
Possible Answer Value
Possible Answer Description
Reference
Dependent entity

Industry and **Organization** have similar usage and therefore can be combined. Also, a number of the survey entities can be combined. The subtypes **Pending Survey**, **Active Survey**, and **Retired Survey** can be rolled up into **Survey**. **Survey Section** is dependent on **Survey** and can be rolled up as well. In addition, because **Survey Category** and **Survey** have similar volatility, they can be combined.

Survey Respondent and **Completed Survey** are close to a one-to-one relationship; there appears to be little if any way to identify when the same person completes more than one survey because we know very little about the respondent. Also, most likely both **Survey Respondent** and **Completed Survey** experience little if any changes over time and therefore have similar volatility.

Survey Question Possible Answer is dependent upon **Possible Answer** and **Survey Question**, yet because there is another relationship from **Survey Question**, we can leave **Survey Question** intact and embed **Possible Answer** into **Survey Question Possible Answer**.

After applying the consolidation above and physical naming standards to the model, we have this physical data model:

Notice there are different ways of combining the entities. Normally when combining, we collapse the entity on the one side (the parent entity) into the entity on the many side (the child). For example, we collapsed **Survey Respondent** into **Completed Survey**, **Possible Answer** into **Survey Question Possible Answer**, and **Survey Category** into **Survey**.

We can also roll up the child entity into the parent entity, as we did with the subtypes as well as with **Survey Section**. We needed to add a **Survey Type Code** which has values *P* for *Pending Survey*, *A* for *Active Survey*, and *R* for *Retired Survey*.

We rolled **Industry** down into **Organization**, forcing us to come up with a maximum number of **Industries** an **Organization** can belong to (in this case three). In MongoDB however, due it its flexible structure, we can have any number of industries in the **Organization** collection by using an array.

STEP 2: ACCOMMODATE HISTORY

Recall there are four types of Slowly Changing Dimensions (SCDs for short). An

SCD of Type 0 means we are only storing the original state and not storing changes. A Type 1 means the most current view. A Type 2 means we are storing all possible changes to the data in a collection—a complete history whenever anything changes. Type 3 means we have a requirement for some history—for example, the most current view and the previous view or the most current view and the original view.

We complete the following Collection History Template for our Survey model:

Collection History Template

	Store only the original state (Type 0)	Store only the most current state (Type 1)	Store full history (Type 2)	Store some history (Type 3)
Organization		✓		
SurveyCreation		✓		
Survey			✓	
CompletedSurvey	✓			
CompletedSurveyResponse	✓			
SurveyQuestion		✓		
SurveyQuestionPossibleAnswer		✓		

The only collection we learned that needs full history is **Survey**. Most collections are Type 1, meaning only the most current. **CompletedSurvey** and

CompletedSurveyResponse are Type 0 (as many transactions are), meaning we are just storing the original state as the transaction should never change.

Using effective dates to implement a Type 2, here is our updated physical data model:

STEP 3: INDEX AND SHARD

The primary and alternate keys from our logical data model are converted into unique indexes on the physical. We can also add additional indexes for performance, called secondary or non-unique indexes. Partitioning (called "sharding" in MongoDB) is when a collection is split up into two or more parts. Vertical partitioning is when fields are split up, and horizontal is when documents are split up.

On our model there are no immediately apparent opportunities to shard. However, we did identify **surveyQuestionLabelName**, **surveyRespondentLastName** and **surveyRespondentFirstName** as being queried frequently, so we added non-unique indexes to each of these (Note that we added a composite index on both **surveyRespondentLastName** and **surveyRespondentFirstName** as they are

frequently queried together):

STEP 4: REVIEW AND CONFIRM

Previously we identified the person or group responsible for validating the model. Now we need to show them the model and make sure it is correct. Often at this stage after reviewing the model, we go back to Step 1 and make some changes and then show the model again. This iterative cycle continues until the model is agreed upon by the validator and deemed correct so that we can move on to MongoDB implementation.

STEP 5: IMPLEMENT

We're done with our modeling, and now we can start creating documents within each of these collections!

Here is a sample document within each collection:

Organization:

```
{ organizationDUNSNumber : "123456789",
```

Industry : [
{ SICCode : “3829”,
industryName : “Restaurant” },
{ SICCode : “1289”,
industryName : “Retail” }
],
organizationName : “The Publyk House”,
organizationFirstSurveyDate : ISODate(“2014-04-05”)
}

Survey:
{ surveyCode : “M329-A”,
surveyEffectiveDate : ISODate(“2014-04-29”),
surveyTypeCode : “A”,
surveyCategoryCode : “CF”,
surveyCategoryName : “Consumer Feedback”,
surveySectionName : “Dining Experience”,
surveySectionDescription : “This section contains those questions pertaining to the quality of the food and overall dining experience.”
}

SurveyCreation:
{ organizationDUNSNumber : “123456789”,
surveyCode : “M329-A”,
surveyEffectiveDate : ISODate(“2014-04-29”),
}

SurveyQuestion:
{ surveyQuestionNumber : “1”,
surveyCode : “M329-A”,
surveyEffectiveDate : ISODate(“2014-04-29”),
surveyQuestionLabelName : “Were you greeted properly?”,
surveyQuestionDescription : “This question lets the respondent rate the hostess.”,

surveyQuestionSingularResponseIndicator : “Y”
}

SurveyQuestionPossibleAnswer:
{ possibleAnswerValue : “3”,
surveyQuestionNumber : “1”,
possibleAnswerDescription : “Average”
}

CompletedSurvey:
{ surveyCode : “M329-A”,
surveyEffectiveDate : ISODate(“2014-04-29”),
surveyRespondentID : “123456”,
completedSurveyDate : ISODate(“2014-05-04”),
surveyRespondentFirstName : “Robbie”
}

CompletedSurveyResponse:
{ surveyCode : “M329-A”,
surveyEffectiveDate : ISODate(“2014-04-29”),
surveyRespondentID : “123456”,
completedSurveyDate : ISODate(“2014-05-04”),
possibleAnswerValue : “3”,
surveyQuestionNumber : “1”
}

APPENDIX A
Answers to Exercises

EXERCISE 1: LIFE WITHOUT DATA MODELING?

I do a lot of data modeling consulting assignments, and I frequently see issues with those projects that skipped data modeling altogether or did just physical data modeling. Issues take the form of data quality issues, integration issues with other applications, support issues, and often communication issues between the business users/sponsor and IT. The project that skipped modeling might have been done on time and within budget, but the long-term costs can be staggering.

EXERCISE 2: SUBTYPING IN MONGODB

ROLLDOWN

Author:

```
{  authorTaxID : "22-5555555",
   authorFirstName : "Steve",
   authorLastName : "Hoberman"
}
```

PrintVersion:

```
{  titleISBN : "9780977140060",
   authorTaxID : "22-5555555",
   printVersionRetailPrice : 44.95,
   titleName : "Data Modeling Made Simple",
   printVersionWeightAmount : 1.5,
```

subtitleName : “A Practical Guide for Business and IT Professionals”

},

{ titleISBN : “9781935504702”,

authorTaxID : “22-5555555”,

printVersionRetailPrice : 39.95,

titleName : “Data Modeling for MongoDB”,

printVersionWeightAmount : 1,

subtitleName : “Building Well-Designed and Supportable MongoDB Databases”

}

eBook:

{ titleISBN : “9781935504719”,

authorTaxID : “22-5555555”,

eBookRetailPrice : 34.95,

titleName : “Data Modeling for MongoDB”,

eBookDownloadSize : 3,

subtitleName : “Building Well-Designed and Supportable MongoDB Databases”

},

{ titleISBN : “9781935504726”,

authorTaxID : “22-5555555”,

eBookRetailPrice : 9.95,

titleName : “The Best Data Modeling Jokes”,

eBookDownloadSize : 2

}

ROLLUP

Author:

{ authorTaxID : “22-5555555”,

authorFirstName : “Steve”,

authorLastName : “Hoberman”

}

Title:

{ titleISBN : “9780977140060”,

```
  authorTaxID : "22-5555555",
  titleName : "Data Modeling Made Simple",
                    subtitleName : "A Practical Guide for Business and IT Professionals",
  titleRetailPrice : 44.95,
  printVersionWeightAmount : 1.5,
  titleTypeCode : "Print"
},
{ titleISBN : "9781935504702",
  authorTaxID : "22-5555555",
  titleName : "Data Modeling for MongoDB",
                    subtitleName : "Building Well-Designed and Supportable MongoDB Databases",
  titleRetailPrice : 39.95,
  titleTypeCode : "Print"
},
{ titleISBN : "9781935504719",
  authorTaxID : "22-5555555",
  titleName : "Data Modeling for MongoDB",
                    subtitleName : "Building Well-Designed and Supportable MongoDB Databases",
  titleRetailPrice : 34.95,
   eBookDownloadSize : 3,
  titleTypeCode : "eBook"
},
{ titleISBN : "9781935504726",
  authorTaxID : "22-5555555",
  titleName : "The Best Data Modeling Jokes",
  titleRetailPrice : 9.95,
   eBookDownloadSize : 2,
  titleTypeCode : "eBook"
}
```

IDENTITY

Author:

```
{ authorTaxID : "22-5555555",
  authorFirstName : "Steve",
```

```
  authorLastName : “Hoberman”
 }

Title:
 { titleISBN : “9780977140060”,
  authorTaxID : “22-5555555”,
  titleName : “Data Modeling Made Simple”,
                    subtitleName : “A Practical Guide for Business and IT Professionals”,
  titleRetailPrice : 44.95
 },
 { titleISBN : “9781935504702”,
  authorTaxID : “22-5555555”,
  titleName : “Data Modeling for MongoDB”,
                    subtitleName : “Building Well-Designed and Supportable MongoDB Databases”,
  titleRetailPrice : 39.95
 },
 { titleISBN : “9781935504719”,
  authorTaxID : “22-5555555”,
  titleName : “Data Modeling for MongoDB”,
                    subtitleName : “Building Well-Designed and Supportable MongoDB Databases”,
  titleRetailPrice : 34.95
 },
 { titleISBN : “9781935504726”,
  authorTaxID : “22-5555555”,
  titleName : “The Best Data Modeling Jokes”,
  titleRetailPrice : 9.95
 }

PrintVersion:
 { titleISBN : “9780977140060”,
  printVersionWeightAmount : 1.5
 },
 { titleISBN : “9781935504702”,
```

printVersionWeightAmount : 1

}

eBook:

{ titleISBN : “9781935504719”,

eBookDownloadSize : 3

},

{ titleISBN : “9781935504726”,

eBookDownloadSize : 2

}

EXERCISE 3: INTERPRETING QUERIES

db.title.find({ $or : [{ pageCount : { $gt : 200 } }, { publicationYear : { $lt : 2010 } }] })

Title Name	Page Count	Publication Year	Author	Amazon Review
Data Modeling Made Simple	250	2009	Steve Hoberman	4.35

db.title.find({ authorName : { $ne : “Shive” } })

Title Name	Page Count	Publication Year	Author	Amazon Review
Extreme Scoping	300	2013	Larissa Moss	4.5
Business unIntelligence	442	2013	Barry Devlin, PhD	5
	250	2009		4.35

EXERCISE 4: MONGODB FUNCTIONALITY

Here is the statement to insert this data into the **Title** collection:

```
db.Title.insert( [ { titleName : "FruITion",
        pageCount : 100,
        publicationYear : "2010",
        author : "Chris Potts"
        },
{ titleName : "Data Quality Assessment",
        pageCount : 400,
        publicationYear : "2009",
        author : "Arkady Maydanchik"
        },
{ titleName : "Data Modeling Theory and Practice",
        pageCount : 300,
        publicationYear : "2008",
        author : "Graeme Simsion"
        } ] )
```

Here is the statement to remove *FruITion* from the Title collection:

```
db.Title.remove( { titleName : "FruITion" } )
```

Here is the statement to update the page count for *Data Quality Assessment* to *350*:

```
db.Title.update ( { titleName : "Data Quality Assessment" }, { "$set" : { pageCount : 350 } } )
```

To view all of the data, run db.Title.find( ):

{ "_id" : ObjectId("5367cdc79e6bbb07630a3a7a"), "titleName" : "Data Quality Assessment", "pageCount" : 350, "publicationYear" : "2009", "author" : "Arkady Maydanchik" }

{ "_id" : ObjectId("5367cdc79e6bbb07630a3a7b"), "titleName" : "Data Modeling Theory and Practice", "pageCount" : 300, "publicationYear" : "2008", "author" : "Graeme Simsion" }

EXERCISE 5: CONCEPTUAL DATA MODELING MINDSET

Here are some of the conceptual data modeling questions I would ask based upon this document:

- What is an **Order**?
- What is an **Order Line**?
- What is a **Product**?
- Can an **Order** contain more than one **Order Line**?
- Can an **Order Line** belong to more than one **Order**?
- Can an **Order** exist without an **Order Line**?
- Can an **Order Line** exist without an **Order**?
- Can a **Product** be referenced by more than one **Order Line**?
- Can an **Order Line** reference more than one **Product**?
- Can a **Product** exist without an **Order Line**?
- Can an **Order Line** exist without a **Product**?

EXERCISE 6: LOGICAL DATA MODELING MINDSET

Here are some of the questions I would ask based upon this collection during the logical data modeling phase:

- How do you identify an **Order**?
- How do you identify an **Order Line**?
- How do you identify a **Product**?
- What properties describe an **Order**?
- What properties describe an **Order Line**?
- What properties describe a **Product**?
- Is **Order Short Description** always required for an **Order**? How long is it?
- Is **Order Scheduled Delivery Date** always required for an **Order**?
- Is **Order Actual Delivery Date** always required for an **Order**?

- What unit of measure is **Order Weight** in?
- What currency is **Order Total Amount** in?
- Does **Order Status Code** ever change over time? If yes, how frequently does it change?

EXERCISE 7: EMBED OR REFERENCE

For the first scenario, I would embed **Account** into **Customer** because, based on the requirements statement, it appears **Customer** and **Account** are frequently queried together. Also, **Account** has no other relationships, so embedding **Account** into **Customer** should not increase complexity. We would have to confirm this, but there is a good chance that a **Customer** owns, on average, just a few **Accounts** and therefore there should be minimal redundancy as well.

For the second scenario, because we are focusing on getting the data in to the system (data entry) instead of querying the system, we would like to minimize redundancy. Therefore I would not embed **Order Line** into **Order**. I am further confident in this decision to keep them separate based on the large size of both concepts and the high cardinality relationship between them (an **Order** on average contains 20 **Order Lines**).

APPENDIX B

References

Chodorow, Kristina. MongoDB: The Definitive Guide. 2nd Edition, O'Reilly Media, Inc., 2013. ISBN 9781449344689.

DAMA International. The DAMA Dictionary of Data Management, 2nd Edition: Over 2,000 Terms Defined for IT and Business Professionals. Technics Publications, LLC, 2011. ISBN 9781935504122.

Hoberman, Steve, and Burbank, Donna and Bradley, Chris. Data Modeling For the Business: A Handbook for Aligning the Business with IT using High-Level Data Models. Technics Publications, LLC, 2009. ISBN 9780977140077.

Hoberman, Steve. Data Modeling Made Simple 2nd Edition: A Practical Guide for Business & Information Technology Professionals. Technics Publications, LLC, 2009. ISBN 9780977140060.

Hoberman, Steve. Data Modeling Made Simple with Embarcadero Data Architect. Technics Publications, LLC, 2013. ISBN 9781935504481.

Kent, William. Data and Reality: A Timeless Perspective on Perceiving and Managing Information in Our Imprecise World. 3rd Edition, Technics Publications, LLC, 2012. ISBN 9781935504214.

The MongoDB 2.4 Manual: http://docs.mongodb.org/manual/#.

The MongoDB Installation Guide: http://www.mongodb.org/downloads.

APPENDIX C
Glossary

abstraction

Abstraction brings flexibility to your data models by redefining and combining some of the attributes, entities, and relationships within the model into more generic terms. For example, we may abstract **Employee** and **Consumer** into the more generic concept of **Person**. A **Person** can play many **Roles**, two of which are **Employee** and **Consumer**.

ACID

ACID stands for Atomic, Consistent, Isolated, and Durable. Atomic means everything within a transaction succeeds or the entire transaction is rolled back. Consistent means that the data accurately reflects any changes up to a certain point in time. A transaction cannot leave the database in an inconsistent state. Isolated means transactions cannot interfere with each other. That is, transactions are independent. Durable means completed transactions persist even when servers restart or there are power failures.

alternate key

An alternate key is a candidate key that, although unique, was not chosen as the primary key but still can be used to find specific entity instances.

architect	An architect is an experienced and skilled designer responsible for system and/or data architecture supporting a broad scope of requirements over time beyond the scope of a single project.
associative entity	An associative entity is an entity that resolves a many-to-many relationship.
attribute	Also known as a "data element," an attribute is a property of importance to the business. Its values contribute to identifying, describing, or measuring instances of an entity. The attribute **Claim Number** identifies each claim. The attribute **Student Last Name** describes the last name of each student. The attribute **Gross Sales Amount** measures the monetary value of a transaction.
BASE	BASE is short for Basically Available, Soft-state, Eventual consistency. Basically Available means there is a response to every query, but that response could be a response saying there was a failure in getting the data, or the response that comes back may be in an inconsistent or changing state. Soft-state means the NoSQL database plays "catch up," updating the database continuously even with changes that occurred from earlier in the day. Eventual consistency means that the system will eventually become consistent once it stops receiving input.
business analyst	A business analyst is an IT or business professional responsible for understanding the business processes and the information needs of an organization, for serving as a liaison between IT and business units, and acting as a facilitator of organizational and cultural change.
candidate key	A candidate key is one or more attributes that uniquely identify an entity instance.

cardinality	Cardinality defines the number of instances of each entity that can participate in a relationship. It is represented by the symbols that appear on both ends of a relationship line.
classword	A classword is the last term in an attribute name such as **Amount**, **Code**, and **Name**. Classwords allow for the assignment of common domains.
collection	A collection is a set of one or more documents. If we had a million orders, we could store all of these order documents in one **Order** collection.
column-oriented	Column-oriented databases such as Cassandra, are NoSQL databases which can work with very complex data types such as unformatted text and imagery, and this data can also be defined on the fly.
concept	A concept is a key idea that is both *basic* and *critical* to your audience. "Basic" means this term is probably mentioned many times a day in conversations with the people who are the audience for the model, which includes the people who need to validate the model as well as the people who need to use the model. "Critical" means the business would be very different or non-existent without this concept.
conceptual data model (CDM)	A conceptual data model is a set of symbols and text representing the key concepts and rules binding these key concepts for a specific business or application scope. The CDM represents the high level business solution.

conformed dimension	A conformed dimension is one that is shared across the business intelligence environment. **Customer**, **Account**, **Employee**, **Product**, **Time**, and **Geography** are examples of conformed dimensions.
data model	A data model is a set of symbols and text which precisely explains a business information landscape. A box with the word "Customer" within it represents the concept of a real **Customer**, such as *Bob*, *IBM*, or *Walmart*, on a data model. A line represents a relationship between two concepts such as capturing that a **Customer** may own one or many **Accounts**.
data modeler	A data modeler is one who confirms and documents data requirements. This role performs the data modeling process.
data modeling	Data modeling is the process of learning about the data, and regardless of technology, this process must be performed for a successful application.
database administrator (DBA)	The DBA is the data professional role responsible for database administration and the function of managing the physical aspects of data resources including database design and integrity, backup and recovery, performance and tuning.
denormalization	Denormalization is the process of selectively violating normalization rules and reintroducing redundancy into the model (and therefore the database). This extra redundancy can reduce data retrieval time, which is the primary reason for denormalizing. We can also denormalize to create a more user-friendly model.
developer	A developer is a person who designs, codes and/or tests software. Synonymous with software developer, systems developer,

	application developer, software engineer, and application engineer.
dimension	A dimension is a subject area whose purpose is to add meaning to the measures. All of the different ways of filtering, sorting, and summing measures make use of dimensions. Dimensions are often, but not exclusively, hierarchies.
dimensional model	A dimensional model (also called "dimensional data model") focuses on easily answering business questions such as "What is our **Gross Sales Amount** by day, product, and region?" Dimensional models are built for the three S's of *Simplicity*, *Speed*, and *Security*.
document	A set of somewhat related data often viewed together. The MongoDB document is analogous to the concept of a record in a relational database.
document-oriented	Document-oriented databases frequently store the business subject in one structure called a "document." For example, instead of storing title and author information in two distinct relational structures, title, author, and other title-related information can all be stored in a single document called **Title**. Document-oriented is much more application focused, as opposed to table oriented which is more data focused. MongoDB is a document-based database.
domain	A domain is the complete set of all possible values that an attribute may be assigned. A domain is a set of validation criteria that can be applied to more than one attribute.
entity	An entity represents a collection of information about something that the business deems important and worthy of capture. A noun or noun phrase identifies a specific entity. It fits into one of several

	categories: who, what, when, where, why, or how.
entity instance	Entity instances are the occurrences or values of a particular entity. The entity **Customer** may have multiple customer instances with names Bob, Joe, Jane, and so forth. The entity **Account** can have instances of Bob's checking account, Bob's savings account, Joe's brokerage account, and so on.
field	The concept of a physical attribute (also called a column or field) in relational databases is equivalent to the concept of a field in MongoDB. MongoDB fields contain two parts, a field name and a field value.
foreign key	A foreign key is an attribute that provides a link to another entity. A foreign key allows a database management system to navigate from one entity to another.
forward engineer	The process of building a new application by starting from the conceptual data model and ending with a database.
grain	The grain is the lowest level of detail available in the meter on a dimensional data model.
graph	A graph database is a NoSQL database designed for data whose relations are well represented as a set of nodes with an undetermined number of connections between those nodes. Graph databases are ideal for capturing social relations (where nodes are people), public transport links (where nodes could be bus or train stations), or road maps (where nodes could be street intersections or highway exits).
	An index is a pointer to something that needs to be retrieved. The

index	index points directly to the place on the disk where the data is stored, thus reducing retrieval time. Indexes work best on attributes whose values are requested frequently but rarely updated.
key-value	A key-value database is a NoSQL database that allows the application to store its data in only two columns ("key" and "value"), with more complex information sometimes stored within the "value" columns.
logical data model (LDM)	A logical data model (LDM) is the detailed business solution to a business problem. It is how the modeler captures the business requirements without complicating the model with implementation concerns such as software and hardware.
measure	A measure is an attribute in a dimensional data model's meter that helps answer one or more business questions.
metadata	Metadata is text, voice, or image that describes what the audience wants or needs to see or experience. The audience could be a person, group, or software program.
meter	A meter is an entity containing a related set of measures. It is a bucket of common measures. As a group, common measures address a business concern such as **Profitability**, **Employee Satisfaction**, or **Sales**.
natural key	Also known as a business key, a natural key is what the business sees as the unique identifier for an entity.
	NoSQL is a name for the category of databases built on non-relational technology. NoSQL is not a good name for what it

NoSQL	represents as it is less about how to query the database (which is where SQL comes in) and more about how the data is stored (which is where relational structures comes in).
normalization	Normalization is the process of applying a set of rules with the goal of organizing *something*. With respect to attributes, normalization ensures that every attribute is single valued and provides a fact completely and only about its primary key.
object	Object includes any data model component such as entities, attributes, and relationships. Objects also include any MongoDB component such as fields, documents, and collections.
partition	Also known as sharding in MongoDB, a partition is a structure that divides or separates. Specific to the physical design, partitioning is used to break a table into rows, columns or both. There are two types of partitioning – vertical and horizontal. Vertical partitioning means separating the columns (the attributes) into separate tables. Horizontal means separating rows (the entity instances) into separate tables.
physical data model (PDM)	The physical data model (PDM) represents the detailed technical solution. The PDM is the logical data model modified for a specific set of software or hardware. The PDM often gives up perfection for practicality, factoring in real concerns such as speed, space, and security.
primary key	A primary key is a candidate key that has been chosen to be *the* unique identifier for an entity.
	A program is a large, centrally organized initiative that contains multiple projects. It has a start date and, if successful, no end date.

program	Programs can be very complex and require long-term modeling assignments. Examples include a data warehouse and a customer relationship management system.
project	A project is a plan to complete a software development effort, often defined by a set of deliverables with due dates. Examples include a sales data mart, broker trading application, reservations system, and an enhancement to an existing application.
recursive relationship	A recursive relationship is a relationship between instances of the same entity. For instance, one organization can report to another organization.
Relational Database Management System	The Relational Database Management System represents the traditional relational database invented by E. F. Codd at IBM in 1970 and first commercially available in 1979 (which was Oracle) [Wikipedia].
relational model	A relational model (also called "relational data model") captures how the business works and contains business rules such as "A **Customer** must have at least one **Account**" or "A **Product** must have a **Product Short Name**."
relationship	Rules are captured on our data model through relationships. A relationship is displayed as a line connecting two entities.
reverse engineer	The process of understanding an existing application by starting with its database and working up through the modeling levels until a conceptual data model is built.
	A secondary key is one or more data elements (if more than one

secondary key	data element, it is called a composite secondary key) that are accessed frequently and need to be retrieved quickly. A secondary key does not have to be unique, or stable, or always contain a value.
slowly changing dimension (SCD)	A Slowly Changing Dimension (SCD) is a term for any reference entity where we need to consider how to handle data changes. There are four basic ways to manage history. An SCD of Type 0 means we are only interested in the original state, an SCD of Type 1 means only the most current state, an SCD of Type 2 means the most current along with all history, and an SCD of Type 3 means the most current and some history will be stored.
spreadsheet	A spreadsheet is a representation of a paper worksheet containing a grid defined by rows and columns, where each cell in the grid can contain text or numbers. The columns often contain different types of information.
stakeholder	A stakeholder is a person who has an interest in the successful completion of a project.
star schema	A star schema is the most common physical dimensional data model structure. A star schema results when each set of structures that make up a dimension is flattened into a single structure. The fact table is in the center of the model and each of the dimensions relate to the fact table at the lowest level of detail.
subject matter expert (SME)	A person with significant experience and knowledge of a given topic or function.
surrogate key	A surrogate key is a primary key that substitutes for a natural key, which is what the business sees as the unique identifier for an entity. It has no embedded intelligence and is used by IT (and not

the business) for integration or performance reasons.

user	A user is a person who enters information into an application or queries the application to answer business questions and produce reports.
view	A view is a virtual table. It is a dynamic “view” or window into one or more tables (or other views) where the actual data is stored.

Index

Footnotes

[1] Note that MongoDB allows transactions to be bundled together so that all of them occur or none of them occur. For instance, we can change an employee's salary and phone number in one statement, ensuring either both occur or none occur. We will cover these statements in Chapter 4.

[2] Note that consistency for NoSQL can have a second definition, which is ensuring that physically distinct data copies are equivalent to each other.

[3] A overview to ISO dates can be found here: http://en.wikipedia.org/wiki/ISO_8601.

[4] Note that if you are using the Class Diagram in the Unified Modeling Language (UML for short), you can specify exact numbers in cardinality.

[5] Ask Asya: http://askasya.com/post/largeembeddedarrays

[6] An important consideration in MongoDB physical modeling is that until version 2.6, MongoDB did not support index intersection, meaning that it would use at most one index per query.