Microsoft

T-SQL Fundamentals

Fourth Edition

Professional

Itzik Ben-Gan

Microsoft

T-SQL Fundamentals

Itzik Ben-Gan

T-SQL Fundamentals

Published with the authorization of Microsoft Corporation by:

Pearson Education, Inc.

ISBN-13: 978-0-13-810210-4

ISBN-10: 0-13-810210-4

Library of Congress Control Number: 2023930537

ScoutAutomatedPrintCode

Editor-in-Chief
Brett Bartow

Executive Editor
Loretta Yates

Associate Editor
Charvi Arora

Development Editor
Songlin Qiu

Managing Editor
Sandra Schroeder

Senior Project Editor
Tracey Croom

Copy Editor
Scout Festa

Indexer
Erika Milllen

Proofreader
Jen Hinchliffe

Technical Editor
Lilach Ben-Gan

Editorial Assistant
Cindy Teeters

Cover Designer
Twist Creative, Seattle

Compositor
codeMantra

Pearson's Commitment to Diversity, Equity, and Inclusion

To Dato,
To live in hearts we leave behind,
Is not to die.

—Thomas Campbell

Contents at a Glance

Contents

Acknowledgments

A number of people contributed to making this book a reality, either directly or indirectly, and deserve thanks and recognition. It's certainly possible I omitted some names unintentionally, and I apologize for this ahead of time.

To Lilach: You're the one who makes me want to be good at what I do. Besides being my inspiration in life, you always take an active role in my books, helping to review the text for the first time. In this book, you took a more official technical editing role, and I can't appreciate enough the errors you spotted, and the many ideas and suggestions for improvements.

To my siblings, Mickey and Ina: Thank you for the constant support and for accepting the fact that I'm away.

To Davide Mauri, Herbert Albert, Gianluca Hotz, and Dejan Sarka: Thanks for your valuable advice when I reached out asking for it.

To the editorial team at Pearson and related vendors. Loretta Yates, many thanks for being so good at what you do and for your positive attitude! Thanks to Charvi Arora for all your hard work and effort. Also, thanks to Songlin Qiu, Scout Festa, Karthik Orukaimani, and Tracey Croom for sifting through all the text and making sure it's polished.

To my friends from Lucient, Fernando G. Guerrero, Herbert Albert, Fritz Lechnitz, and many others. We've been working together for over two decades, and it's been quite a ride!

To members of the Microsoft SQL Server development team, Umachandar Jayachandran (UC), Conor Cunningham, Kevin Farlee, Craig Freedman, Kendal Van Dyke, Derek Wilson, Davide Mauri, Bob Ward, Buck Woody, and I'm sure many others. Thanks for creating such a great product, and thanks for all the time you spent meeting with me and responding to my emails, addressing my questions, and answering my requests for clarification.

To Aaron Bertrand, who besides being one of the most active and prolific SQL Server pros I know, does an amazing job editing the sqlperformance.com content, including my articles.

To Data Platform MVPs, past and present: Erland Sommarskog, Aaron Bertrand, Hugo Kornelis, Paul White, Alejandro Mesa, Tibor Karaszi, Simon Sabin, Denis Reznik, Tony Rogerson, and many others—and to the Data Platform MVP lead, Rie Merritt. This is a great program that I'm grateful for and proud to be part of. The level of expertise of this group is amazing, and I'm always excited when we all get to meet, both to share ideas and just to catch up at a personal level.

Finally, to my students: Teaching about T-SQL is what drives me. It's my passion. Thanks for allowing me to fulfill my calling and for all the great questions that make me seek more knowledge.

About the Author

ITZIK BEN-GAN is a leading authority on T-SQL, regularly teaching, lecturing, and writing on the subject. He has delivered numerous training events around the world focused on T-SQL Querying, Query Tuning, and Programming. He is the author of several books including *T-SQL Fundamentals*, *T-SQL Querying*, and *T-SQL Window Functions*. Itzik has been a Microsoft Data Platform MVP (Most Valuable Professional) since 1999.

Introduction

This book walks you through your first steps in T-SQL (also known as *Transact-SQL*), which is the Microsoft SQL Server dialect of the ISO/IEC and ANSI standards for SQL. You'll learn the theory behind T-SQL querying and programming and how to develop T-SQL code to query and modify data, and you'll get a brief overview of programmable objects.

Although this book is intended for beginners, it's not merely a set of procedures for readers to follow. It goes beyond the syntactical elements of T-SQL and explains the logic behind the language and its elements.

Occasionally, the book covers subjects that might be considered advanced for readers who are new to T-SQL; therefore, you should consider those sections to be optional reading. If you feel comfortable with the material discussed in the book up to that point, you might want to tackle these more advanced subjects; otherwise, feel free to skip those sections and return to them after you gain more experience.

Many aspects of SQL are unique to the language and very different from other programming languages. This book helps you adopt the right state of mind and gain a true understanding of the language elements. You learn how to think in relational terms and follow good SQL programming practices.

The book is not version specific; it does, however, cover language elements that were introduced in recent versions of SQL Server, including SQL Server 2022. When I discuss language elements that were introduced recently, I specify the version in which they were added.

Besides being available as an on-premises, or box, flavor, SQL Server is also available as cloud-based flavors called Azure SQL Database and Azure SQL Managed Instance. The code samples in this book are applicable to both the box and cloud flavors of SQL Server.

To complement the learning experience, the book provides exercises you can use to practice what you learn. I cannot emphasize enough the importance of working on those exercises, so make sure not to skip them!

Who Should Read This Book

This book is intended for T-SQL developers, database administrators (DBAs), business intelligence (BI) practitioners, data scientists, report writers, analysts, architects, and SQL Server power users who just started working with SQL Server and who need to write queries and develop code using T-SQL.

This book covers fundamentals. It's mainly aimed at T-SQL practitioners with little or no experience. With that said, several readers of the previous editions of this book have mentioned that—even though they already had years of experience—they still found the book useful for filling in gaps in their knowledge.

This book assumes that you are familiar with basic concepts of relational database management systems.

Organization of This Book

This book starts with a theoretical background to T-SQL querying and programming in Chapter 1, laying the foundation for the rest of the book, and provides basic coverage of creating tables and defining data integrity. The book covers various aspects of querying and modifying data in Chapters 2 through 8, and holds a discussion of transactions and concurrency in Chapter 10. In Chapter 9 and Chapter 11 the book covers specialized topics including temporal tables and SQL Graph. Finally, the book provides a brief overview of programmable objects in Chapter 12.

Here's a list of the chapters along with a short description of the content in each chapter:

- Chapter 1, "Background to T-SQL querying and programming," provides the theoretical background for SQL, set theory, and predicate logic. It examines relational theory, describes SQL Server's architecture, and explains how to create tables and define data integrity.
- Chapter 2, "Single-table queries," covers various aspects of querying a single table by using the *SELECT* statement.
- Chapter 3, "Joins," covers querying multiple tables by using joins, including cross joins, inner joins, and outer joins.
- Chapter 4, "Subqueries," covers queries within queries, otherwise known as *subqueries*.
- Chapter 5, "Table expressions," covers derived tables, Common Table Expressions (CTEs), views, inline table-valued functions (iTVFs), and the *APPLY* operator.

- Chapter 6, “Set operators,” covers the set operators *UNION*, *INTERSECT*, and *EXCEPT*.
- Chapter 7, “T-SQL for data analysis,” covers window functions, pivoting, unpivoting, working with grouping sets, and handling time-series data.
- Chapter 8, “Data modification,” covers inserting, updating, deleting, and merging data.
- Chapter 9, “Temporal tables,” covers system-versioned temporal tables.
- Chapter 10, “Transactions and concurrency,” covers concurrency of user connections that work with the same data simultaneously; it covers transactions, locks, blocking, isolation levels, and deadlocks.
- Chapter 11, “SQL Graph,” covers modeling data using graph-based concepts such as nodes and edges. It includes creating, modifying, and querying graph-based data.
- Chapter 12, “Programmable objects,” provides a brief overview of the T-SQL programming capabilities in SQL Server.
- The book also provides an appendix, “Getting started,” to help you set up your environment, download the book’s source code, install the *TSQLV6* sample database, start writing code against SQL Server, and learn how to get help by working with the product documentation.

System Requirements

The appendix, “Getting started,” explains which editions of SQL Server 2022 you can use to work with the code samples included with this book. Each edition of SQL Server might have different hardware and software requirements, and those requirements are described in the product documentation, under “Hardware and Software Requirements for Installing SQL Server 2022,” at the following URL: *https://learn.microsoft.com/en-us/sql/sql-server/install/hardware-and-software-requirements-for-installing-sql-server-2022*. The appendix also explains how to work with the product documentation.

If you’re connecting to Azure SQL Database or Azure SQL Managed Instance, hardware and server software are handled by Microsoft, so those requirements are irrelevant in this case.

For the client tool to run the code samples against SQL Server, Azure SQL Database, and Azure SQL Managed Instance, you can use either SQL Server Management Studio (SSMS) or Azure Data Studio (ADS). You can download SSMS at *https://learn.microsoft.com/en-us/sql/ssms*. You can download Azure Data Studio at *https://learn.microsoft.com/en-us/sql/azure-data-studio*.

Code Samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All source code, including exercises and solutions, can be downloaded from the following webpage:

MicrosoftPressStore.com/TSQLFund4e/downloads

Follow the instructions to download the TSQLFundamentalsYYYYMMDD.zip file, where *YYYYMMDD* reflects the last update date of the source code.

Refer to the appendix, "Getting started," for details about the source code.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

MicrosoftPressStore.com/TSQLFund4e/errata

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit *MicrosoftPressStore.com/Support*

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

CHAPTER 1

Background to T-SQL querying and programming

You're about to embark on a journey to a land that is like no other—a land that has its own set of laws. If reading this book is your first step in learning Transact-SQL (T-SQL), you should feel like Alice—just before she started her adventures in Wonderland. For me, the journey has not ended; instead, it's an ongoing path filled with new discoveries. I envy you; some of the most exciting discoveries are still ahead of you!

I've been involved with T-SQL for many years: teaching, speaking, writing, and consulting about it. T-SQL is more than just a language—it's a way of thinking. In my first few books about T-SQL, I've written extensively on advanced topics, and for years I have postponed writing about fundamentals. This is not because T-SQL fundamentals are simple or easy—in fact, it's just the opposite. The apparent simplicity of the language is misleading. I could explain the language syntax elements in a superficial manner and have you writing queries within minutes. But that approach would only hold you back in the long run and make it harder for you to understand the essence of the language.

Acting as your guide while you take your first steps in this realm is a big responsibility. I wanted to make sure that I spent enough time and effort exploring and understanding the language before writing about its fundamentals. T-SQL is deep; learning the fundamentals the right way involves much more than just understanding the syntax elements and coding a query that returns the right output. You need to forget what you know about other programming languages and start thinking in terms of T-SQL.

Theoretical background

SQL stands for *Structured Query Language*. SQL is a standard language that was designed to query and manage data in relational database management systems (RDBMSs). An RDBMS is a database management system based on the relational model (a semantic model for representing data), which in turn is based on two mathematical branches: set theory and predicate logic. Many other programming languages and various aspects of computing evolved pretty much as a result of intuition. In contrast, to the degree that SQL is based on the relational model, it is based on a firm foundation—applied mathematics. T-SQL thus sits on wide and solid shoulders. Microsoft provides T-SQL as a dialect of, or an extension to, SQL in SQL Server—its on-premises RDBMS flavor, and in Azure SQL and Azure Synapse Analytics—its cloud-based RDBMS flavors.

Note The term *Azure SQL* collectively refers to three different cloud offerings: Azure SQL Database, Azure SQL Managed Instance, and SQL Server on Azure VM. I describe the differences between these offerings later in the chapter.

This section provides a brief theoretical background about SQL, set theory and predicate logic, the relational model, and types of database systems. Because this book is neither a mathematics book nor a design/data-modeling book, the theoretical information provided here is informal and by no means complete. The goals are to give you a context for the T-SQL language and to deliver the key points that are integral to correctly understanding T-SQL later in the book.

Language independence

The relational model is language independent. That is, you can apply data management and manipulation following the relational model's principles with languages other than SQL—for example, with C# in an object model. Today it is common to see RDBMSs that support languages other than just a dialect of SQL—for example, the integration of the CLR, Java, Python, and R in SQL Server, with which you can handle tasks that historically you handled mainly with SQL, such as data manipulation.

Also, you should realize from the start that SQL deviates from the relational model in several ways. Some even say that a new language—one that more closely follows the relational model—should replace SQL. But to date, SQL is the de facto language used by virtually all leading RDBMSs.

See Also For details about the deviations of SQL from the relational model, as well as how to use SQL in a relational way, see this book on the topic: *SQL and Relational Theory: How to Write Accurate SQL Code, 3rd Edition,* by C. J. Date (O'Reilly Media, 2015).

SQL

SQL is both an ANSI and ISO standard language based on the relational model, designed for querying and managing data in an RDBMS.

In the early 1970s, IBM developed a language called SEQUEL (short for Structured English QUEry Language) for its RDBMS product called System R. The name of the language was later changed from SEQUEL to SQL because of a trademark dispute. SQL first became an ANSI standard in 1986, and then an ISO standard in 1987. Since 1986, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) have been releasing revisions for the SQL standard every few years. So far, the following standards have been released: SQL-86 (1986), SQL-89 (1989), SQL-92

(1992), SQL:1999 (1999), SQL:2003 (2003), SQL:2006 (2006), SQL:2008 (2008), SQL:2011 (2011), and SQL:2016 (2016). The SQL standard is made of multiple parts. Part 1 provides the framework and Part 2 defines the foundation with the core SQL elements. The other parts define standard extensions, such as SQL for XML, SQL-Java integration, and others.

Interestingly, SQL resembles English and is also very logical. Unlike many programming languages, which use an imperative programming paradigm, SQL uses a declarative one. That is, SQL requires you to specify *what* you want to get and not *how* to get it, letting the RDBMS figure out the physical mechanics required to process your request.

SQL has several categories of statements, including data definition language (DDL), data manipulation language (DML), and data control language (DCL). DDL deals with object definitions and includes statements such as *CREATE, ALTER,* and *DROP*. DML allows you to query and modify data and includes statements such as *SELECT, INSERT, UPDATE, DELETE, TRUNCATE,* and *MERGE*. It's a common misunderstanding that DML includes only data-modification statements, but as I mentioned, it also includes *SELECT*. Another common misunderstanding is that *TRUNCATE* is a DDL statement, but in fact it is a DML statement. DCL deals with permissions and includes statements such as *GRANT* and *REVOKE*. This book focuses on DML.

T-SQL is based on standard SQL, but it also provides some nonstandard/proprietary extensions. Moreover, T-SQL does not implement all of standard SQL. When describing a language element for the first time, I'll typically mention if it's nonstandard.

Set theory

Set theory, which originated with the mathematician Georg Cantor, is one of the mathematical branches on which the relational model is based. Cantor's definition of a set follows:

> *By a "set" we mean any collection M into a whole of definite, distinct objects m (which are called the "elements" of M) of our perception or of our thought.*
>
> —GEORG CANTOR: HIS MATHEMATICS AND PHILOSOPHY OF THE INFINITE, *BY JOSEPH W. DAUBEN (PRINCETON UNIVERSITY PRESS, 2020)*

Every word in the definition has a deep and crucial meaning. The definitions of a set and set membership are axioms that are not supported by proofs. Each element belongs to a universe, and either is or is not a member of the set.

Let's start with the word *whole* in Cantor's definition. A set should be considered a single entity. Your focus should be on the collection of objects as opposed to the individual objects that make up the collection. Later on, when you write T-SQL queries against tables in a database (such as a table of employees), you should think of the set of employees as a whole rather than the individual employees.

This might sound trivial and simple enough, but apparently many programmers have difficulty adopting this way of thinking.

The word *distinct* means that every element of a set must be unique. Jumping ahead to tables in a database, you can enforce the uniqueness of rows in a table by defining key constraints. Without a key, you won't be able to uniquely identify rows, and therefore the table won't qualify as a set. Rather, the table would be a *multiset* or a *bag*.

The phrase *of our perception or of our thought* implies that the definition of a set is subjective. Consider a classroom: one person might perceive a set of people, whereas another might perceive a set of students and a set of teachers. Therefore, you have a substantial amount of freedom in defining sets. When you design a data model for your database, the design process should carefully consider the subjective needs of the application to determine adequate definitions for the entities involved.

As for the word *object*, the definition of a set is not restricted to physical objects, such as cars or employees, but rather is relevant to abstract objects as well, such as prime numbers or lines.

What Cantor's definition of a set leaves out is probably as important as what it includes. Notice that the definition doesn't mention any order among the set elements. The order in which set elements are listed is not important. The formal notation for listing set elements uses curly brackets: {a, b, c}. Because order has no relevance, you can express the same set as *{b, a, c}* or *{b, c, a}*. Jumping ahead to the set of attributes (*columns* in SQL) that make up the heading of a relation (*table* in SQL), an element (in this case, an attribute) is supposed to be identified by name—not by ordinal position.

Similarly, consider the set of tuples (*rows* in SQL) that make up the body of the relation; an element (in this case a tuple) is identified by its key values—not by position. Many programmers have a hard time adapting to the idea that, with respect to querying tables, there is no order among the rows. In other words, a query against a table can return table rows in *any order* unless you explicitly request that the data be ordered in a specific way, perhaps for presentation purposes.

Predicate logic

Predicate logic, whose roots go back to ancient Greece, is another branch of mathematics on which the relational model is based. Dr. Edgar F. Codd, in creating the relational model, had the insight to connect predicate logic to both the management and querying of data. Loosely speaking, a *predicate* is a property or an expression that either holds or doesn't hold—in other words, is either true or false. The relational model relies on predicates to maintain the logical integrity of the data and define its structure. One example of a predicate used to enforce integrity is a constraint defined in a table called *Employees* that allows only employees with a salary greater than zero to be stored in the table. The predicate is "salary greater than zero" (T-SQL expression: *salary > 0*).

You can also use predicates when filtering data to define subsets, and more. For example, if you need to query the *Employees* table and return only rows for employees from the sales department, you use the predicate "department equals sales" in your query filter (T-SQL expression: *department = 'sales'*).

In set theory, you can use predicates to define sets. This is helpful because you can't always define a set by listing all its elements (for example, infinite sets), and sometimes for brevity it's more convenient to define a set based on a property. As an example of an infinite set defined with a predicate, the set of all prime numbers can be defined with the following predicate: "*x* is a positive integer greater than 1 that is divisible only by 1 and itself." For any specified value, the predicate is either true or not true. The set of all prime numbers is the set of all elements for which the predicate is true. As an example of a finite set defined with a predicate, the set *{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}* can be defined as the set of all elements for which the following predicate holds true: "*x* is an integer greater than or equal to 0 and smaller than or equal to 9."

The relational model

The relational model is a semantic model for data management and manipulation and is based on set theory and predicate logic. As mentioned earlier, it was created by Dr. Edgar F. Codd, and later explained and developed by Chris Date, Hugh Darwen, and others. The first version of the relational model was proposed by Codd in 1969 in an IBM research report called "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks." A revised version was proposed by Codd in 1970 in a paper called "A Relational Model of Data for Large Shared Data Banks," published in the journal *Communications of the ACM*.

The goal of the relational model is to enable consistent representation of data with minimal or no redundancy and without sacrificing completeness, and to define data integrity (enforcement of data consistency) as part of the model. An RDBMS is supposed to implement the relational model and provide the means to store, manage, enforce the integrity of, and query data. The fact that the relational model is based on a strong mathematical foundation means that given a certain data-model instance (from which a physical database will later be generated), you can tell with certainty when a design is flawed, rather than relying solely on intuition.

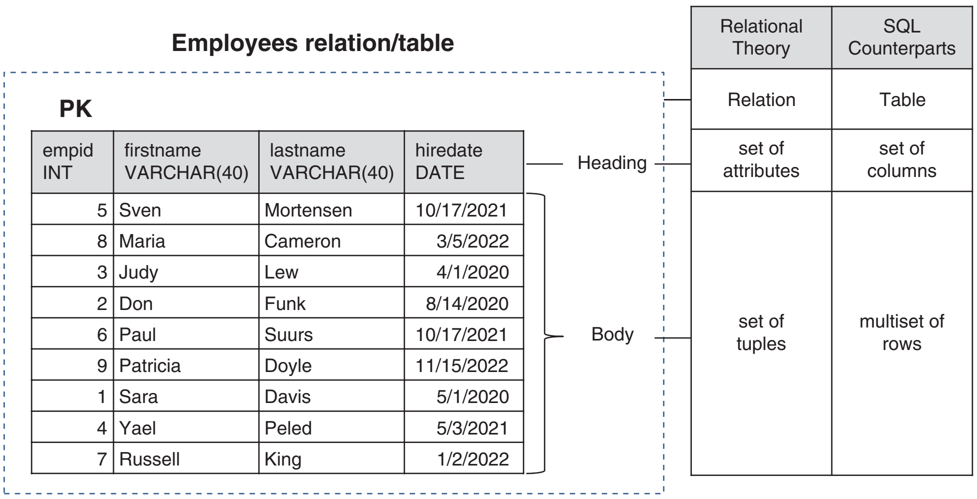
The relational model involves concepts such as propositions, predicates, relations, tuples, attributes, and more. For nonmathematicians, these concepts can be quite intimidating. The sections that follow cover some key aspects of the model in an informal, nonmathematical manner and explain how they relate to databases.

Propositions, predicates, and relations

The common belief that the term *relational* stems from relationships between tables is incorrect. "Relational" actually pertains to the mathematical term *relation*. In set theory, a relation is a representation of a set. In the relational model, a relation is a set of related information, with the counterpart in SQL being a table—albeit not an exact counterpart. A key point in the relational model is that a single relation should represent a single set (for example, *Customers*). Note that operations on relations (based on relational algebra) result in a relation (for example, an intersection between two relations). This is what's known as the *closure* property of the relational algebra, and is what enables the nesting of relational expressions.

Note The relational model distinguishes between a *relation* and a *relation variable*, but to keep things simple, I won't get into this distinction. Instead, I'll use the term *relation* for both cases. Also, as Figure 1-1 shows, a relation is made of a heading and a body. The heading consists of a set of attributes (*columns* in SQL), where each element has a name and a type name and is identified by name. The body consists of a set of tuples (*rows* in SQL), where each element is identified by a key. To keep things simple, I'll often refer to a table as a set of rows.

Figure 1-1 shows an illustration of a relation called Employees. It compares the components of a relation in relational theory with those of a table in SQL.



empid INT	firstname VARCHAR(40)	lastname VARCHAR(40)	hiredate DATE
5	Sven	Mortensen	10/17/2021
8	Maria	Cameron	3/5/2022
3	Judy	Lew	4/1/2020
2	Don	Funk	8/14/2020
6	Paul	Suurs	10/17/2021
9	Patricia	Doyle	11/15/2022
1	Sara	Davis	5/1/2020
4	Yael	Peled	5/3/2021
7	Russell	King	1/2/2022

Relational Theory	SQL Counterparts
Relation	Table
set of attributes	set of columns
set of tuples	multiset of rows

FIGURE 1-1 Illustration of Employees relation

Be aware that creating a truly adequate visual representation of a relation is very difficult in practice, since the set of attributes making the heading of a relation has no order, and the same goes for the set of tuples making the body of a relation. In an illustration, it might seem like those elements do have order even though they don't. Just make sure to keep this in mind.

When you design a data model for a database, you represent all data with relations (tables). You start by identifying propositions that you will need to represent in your database. A proposition is an assertion or a statement that must be true or false. For example, the statement, "Employee Jiru Ben-Gan was born on June 22, 2003, and works in the Pet Food department" is a proposition. If this proposition is true, it will manifest itself as a row in a table of *Employees*. A false proposition simply won't manifest itself. This presumption is known as the *closed-world assumption (CWA)*.

The next step is to formalize the propositions. You do this by taking out the actual data (the body of the relation) and defining the structure (the heading of the relation)—for example, by creating predicates out of propositions. You can think of predicates as parameterized propositions. The heading of a relation comprises a set of attributes. Note the use of the term "set"; in the relational model, attributes are unordered and distinct. An attribute has a name and a type name, and is identified by name. For example, the heading of an *Employees* relation might consist of the following attributes (expressed as pairs of attribute names and type names): *employeeid* integer, *firstname* character string, *lastname* character string, *birthdate* date, and *departmentid* integer.

A *type* is one of the most fundamental building blocks for relations. A type constrains an attribute to a certain set of possible or valid values. For example, the type *INT* is the set of all integers in the range –2,147,483,648 to 2,147,483,647. A type is one of the simplest forms of a predicate in a database because it restricts the attribute values that are allowed. For example, the database would not accept a proposition where an employee birth date is February 31, 2003 (not to mention a birth date stated as something like "abc!"). Note that types are not restricted to base types such as integers or dates; a type can also be an enumeration of possible values, such as an enumeration of possible job positions. A type can be simple or complex. Probably the best way to think of a type is as a class—encapsulated data and the behavior supporting it. An example of a complex type is a geometry type that supports polygons.

Missing values

There's an aspect of the relational model and SQL that is the source of many passionate debates. Whether to support the notion of missing values and three-valued predicate logic. That is, in two-valued predicate logic, a predicate is either true or false. If a predicate is not true, it must be false. Use of two-valued predicate logic follows a mathematical law called "the law of excluded middle." However, some support the idea of three-valued predicate logic, taking into account cases where values are missing. A predicate involving a missing value yields neither *true* nor *false* as the result truth value—it yields *unknown*.

Take, for example, a *mobilephone* attribute of an *Employees* relation. Suppose that a certain employee's mobile phone number is missing. How do you represent this fact in the database? One option is to have the *mobilephone* attribute allow the use of a special marker for a missing value. Then a predicate used for filtering purposes, comparing the *mobilephone* attribute with some specific number, will yield *unknown* for the case with the missing value. Three-valued predicate logic refers to the three possible truth values that can result from a predicate—*true*, *false*, and *unknown*.

Some people believe that *NULLs* and three-valued predicate logic are nonrelational, whereas others believe that they are relational. Codd actually advocated for four-valued predicate logic, saying that there were two different cases of missing values: missing but applicable (A-Values marker), and missing but inapplicable (I-Values marker). An example of "missing but applicable" is when an employee has a mobile phone, but you don't know what the mobile phone number is. An example of "missing but inapplicable" is when an employee doesn't have a mobile phone at all. According to Codd, two special markers should be used to support these two cases of missing values. SQL doesn't make a distinction between the two cases for missing values that Codd does; rather, it defines the *NULL* marker to signify any kind of missing value. It also supports three-valued predicate logic. Support for *NULLs* and three-valued predicate logic in SQL is the source of a great deal of confusion and complexity, though one can argue that missing values are part of reality. In addition, the alternative—using only two-valued predicate logic and representing missing values with your own custom means—is not necessarily less problematic.

Note As mentioned, a *NULL* is not a value but rather a marker for a missing value. Therefore, though unfortunately it's common, the use of the terminology "*NULL* value" is incorrect. The correct terminology is "*NULL* marker" or just "*NULL*." In the book, I typically use the latter because it's more common in the SQL community.

Constraints

One of the greatest benefits of the relational model is the ability to define data integrity as part of the model. Data integrity is achieved through rules called *constraints* that are defined in the data model and enforced by the RDBMS. The simplest methods of enforcing integrity are assigning an attribute type and "nullability" (whether it supports or doesn't support *NULLs*). Constraints are also enforced through the model itself; for example, the relation *Orders(orderid, orderdate, duedate, shipdate)* allows three distinct dates per order, whereas the relations *Employees(empid)* and *EmployeeChildren(empid, childname)* allow zero to countable infinity children per employee.

Other examples of constraints include the enforcement of *candidate keys,* which provide entity integrity, and *foreign keys,* which provide referential integrity. A candidate key is a key defined on one or more attributes of a relation. Based on a candidate key's attribute values you can uniquely identify a tuple (row). A constraint enforcing a candidate key prevents duplicates. You can identify multiple candidate keys in a relation. For example, in an *Employees* relation, you can have one candidate key based on *employeeid*, another on *SSN* (Social Security number), and others. Typically, you arbitrarily choose one of the candidate keys as the *primary key* (for example, *employeeid* in the *Employees* relation) and use that as the preferred way to identify a row. All other candidate keys are known as *alternate keys*.

Foreign keys are used to enforce referential integrity. A foreign key is defined on one or more attributes of a relation (known as the *referencing relation*) and references a candidate key in another (or possibly the same) relation. This constraint restricts the values in the referencing relation's foreign-key attributes to the values that appear in the referenced relation's candidate-key attributes. For example, suppose that the *Employees* relation has a foreign key defined on the attribute *departmentid*, which references the primary-key attribute *departmentid* in the *Departments* relation. This means that the values in *Employees.departmentid* are restricted to the values that appear in *Departments.departmentid*.

Normalization

The relational model also defines *normalization rules* (also known as *normal forms*). Normalization is a formal mathematical process to guarantee that each entity will be represented by a single relation. In a normalized database, you avoid anomalies during data modification and keep redundancy to a minimum without sacrificing completeness. If you follow entity relationship modeling (ERM) and represent each entity and its attributes, you probably won't need normalization; instead, you will apply normalization only to reinforce and ensure that the model is correct. You can find the definition of ERM in the following Wikipedia article: *https://en.wikipedia.org/wiki/Entity–relationship_model*.

The following sections briefly cover the first three normal forms (1NF, 2NF, and 3NF) introduced by Codd.

1NF

The first normal form says that the tuples (rows) in the relation (table) must be unique and attributes should be atomic. This is a redundant definition of a relation; in other words, if a table truly represents a relation, it is already in first normal form.

You enforce the uniqueness of rows in SQL by defining a primary key or unique constraint in the table.

You can operate on attributes only with operations that are defined as part of the attribute's type. Atomicity of attributes is subjective in the same way that the definition of a set is subjective. As an example, should an employee name in an *Employees* relation be expressed with one attribute (*fullname*), two attributes (*firstname* and *lastname*), or three attributes (*firstname*, *middlename*, and *lastname*)? The answer depends on the application. If the application needs to manipulate the parts of the employee's name separately (such as for search purposes), it makes sense to break them apart; otherwise, it doesn't.

In the same way that an attribute might not be atomic enough based on the needs of the applications that use it, an attribute might also be subatomic. For example, if an address attribute is considered atomic for the applications that use it, not including the city as part of the address would violate the first normal form.

This normal form is often misunderstood. Some people think that an attempt to mimic arrays violates the first normal form. An example would be defining a *YearlySales* relation with the following attributes: *salesperson*, *qty2020*, *qty2021*, and *qty2022*. However, in this example, you don't really violate the first normal form; you simply impose a constraint—restricting the data to three specific years: 2020, 2021, and 2022.

2NF

The second normal form involves two rules. One rule is that the data must meet the first normal form. The other rule addresses the relationship between nonkey and candidate-key attributes. For every candidate key, every nonkey attribute has to be fully functionally dependent on the entire candidate key. In other words, a nonkey attribute cannot be fully functionally dependent on part of a candidate key. To put it more informally, if you need to obtain any nonkey attribute value, you need to provide the values of all attributes of a candidate key from the same tuple. You can find any value of any attribute of any tuple if you know all the attribute values of a candidate key.

As an example of violating the second normal form, suppose that you define a relation called *Orders* that represents information about orders and order lines. (See Figure 1-2.) The *Orders* relation contains the following attributes: *orderid*, *productid*, *orderdate*, *qty*, *customerid*, and *companyname*. The primary key is defined on *orderid* and *productid*.