Chapter 2
BRIEF OVERVIEW OF RELATIONAL MODEL

Introduction

The first chapter provided a foundational understanding of WHY humans capture data (our fundamental desire to be efficient competitors who “learn to win”). We also reviewed how data has been captured throughout history with oral tradition and paper-based systems for the past 40,000 years and then hierarchical systems starting with the first electronic computers post-World War 2. Remember, hierarchical database systems were designed to imitate the paper-based systems we had used for thousands of years but soon became seen as problematic because of bloat from duplicate data being stored in multiple hierarchies. Like an old-school filing cabinet that holds paper, hierarchical data structures arrange important data in perspectives that benefit only a small percentage of queries.

As we saw, not only were hierarchical systems fat with multiple copies of the same data in different structures, the nature of having multiple structures meant that each INSERT, UPDATE, and DELETE transaction had to be duplicated across each hierarchy. The duplicated effort meant each transaction was multiple times longer in duration as each hierarchy was written. Further still, the logic behind maintaining data integrity across multiple structures was complicated; workers often sought to speed up their tasks by keeping secret pockets of data outside the regular hierarchies which led to misinformation and data anomalies.

Everything changed for the better with the definition of the relational model in 1969, when Dr. E.F. Codd, a scientist with IBM, proposed a novel design based on relational theory. In relational theory, each object is now represented as a table, and an entire database becomes a collection of related tables as opposed to multiple hierarchical structures based

on a perspective. Instead of having to duplicate data, relational databases have the concept of 'keys' that join tables together, forming the same wider collection of data with significantly less effort and no redundancy. This design has been effectively unchanged in the 56 years since it was defined in the late 1960s.

The benefits of relational theory are massive! This chapter explores the details and helps each aspiring data scientist understand the 'science' beneath the basic structures many people take for granted. Learnings from this chapter will provide analysts with the ability to become functionally independent from infrastructure engineers, developing their design skills capable of building a robust database in third normal form (3NF) within hours.

There are many dependencies between current and ancient database design principles, basic SQL, data modeling, and more advanced coding aspects of SQL. There is often conflict over which topics to introduce first and how deep to linger before moving on to the next. In short, the focus of this textbook is to provide a solid foundational understanding for those people who want to leverage SQL to become superior data scientists. This textbook will therefore need to jump around a bit when conveying a technical concept with a range of examples that may feel incomplete now, we move on to the next topic. Trust that the material will come together as the journey continues!

Structure

By the end of this chapter, you will:

- Identify the design structure and characteristics of hierarchical databases
- Understand the premise of the Relational Model and its major features
- Explain the proper use of primary keys (PK) and foreign keys (FK)
- Explain cardinality and the use of Crow's Foot notation when defining relationships between multiple entities
- Articulate the range of data types used in relational model with their best uses
- Recognize the basic rules of relational database design principles and be ready to construct an original database of your choosing!

If we are on pace with the material covered so far, we might be a little uncertain about what the big deal about computerized databases is, and perhaps why we are spending time hearing about the past when probably most people want to start coding! To fully develop skills around data science and especially SQL, it helps to understand the design structures that are superior to the paper-based methods from previous generations (if only to avoid being like your grandparents!). The reason is nearly everyone who designs a relational

database for the first time makes the same handful of mistakes! Some of the most brilliant minds on planet Earth have spent time working at Microsoft, Walt Disney Company, Bill and Melinda Gates Foundation, and the Institute for Health Metrics and Evaluation, and even these people make the predictable mistakes we will cover. The better we avoid these common mistakes, the better your future with data will be as you can design robust, quality, structures with speed and confidence.

Back in the mid-20th century, people had modern transportation options (cars, airplanes, trains) as well as television, radio, and telephones for communication purposes. They did not, unfortunately, have the modern conveniences of the internet, cell phones, or electronic databases. As such, even large companies in the 1960s managed their entire business with paper-based systems (hand-written bookkeeping). Please imagine the pace at which businesses operated in the time of typewriters! Events such as registration for classes took days, while ordering material from a catalog often took weeks. It seems many consumers nowadays get irritated when an order takes more than 45 seconds!

Hierarchical Database: Quick Review

Hierarchical databases, as we saw in chapter 1, must read and assemble all data from the underlying file one-by-one and then evaluate whether the record is desirable based on the logic of a query. This process also often ends up casting away more than 99% of records for each query after spending the time reading everything! This design is very inefficient, especially back in the 1960s when a single MEGABYTE of memory (not gigabyte…1/1000th of a GB) cost about the same money as a brand-new car.

Just like in modern times, companies had to be able to answer hundreds of questions. By design, a hierarchy has a top-level object by which all the remaining data is assembled underneath. Let us remind ourselves of one hierarchy each from METRO_TRANSIT and MUSIC_STREAM from chapter 1 '*Organize or Die*':

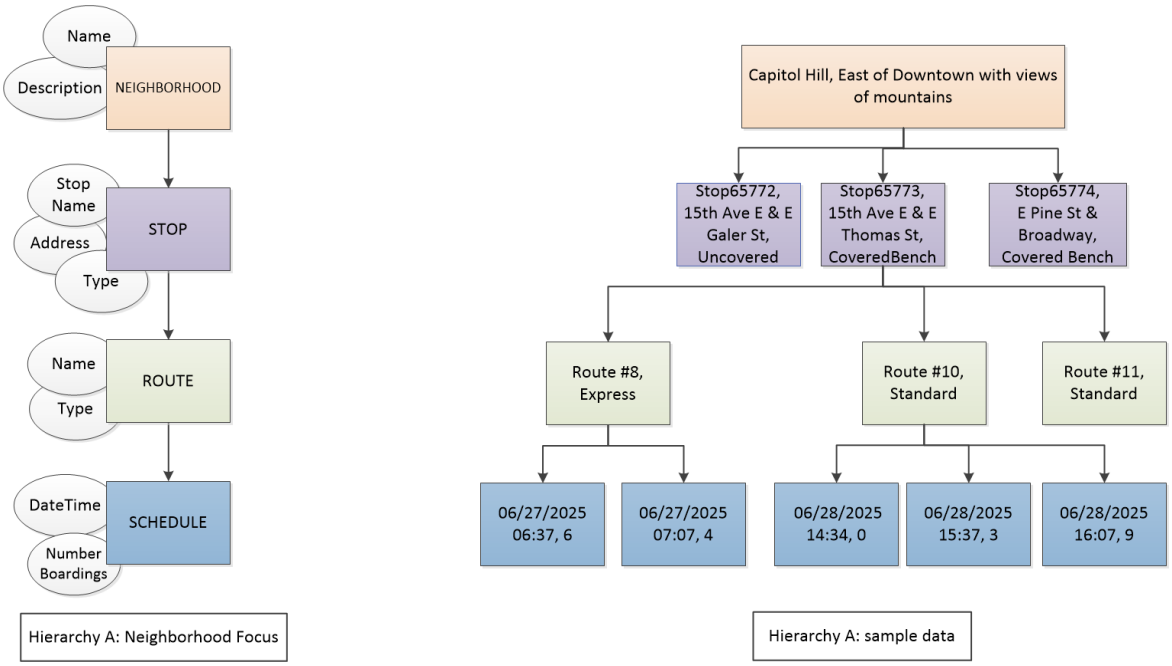

Figure 2.1: Example hierarchy of neighborhood trips for METRO_TRANSIT

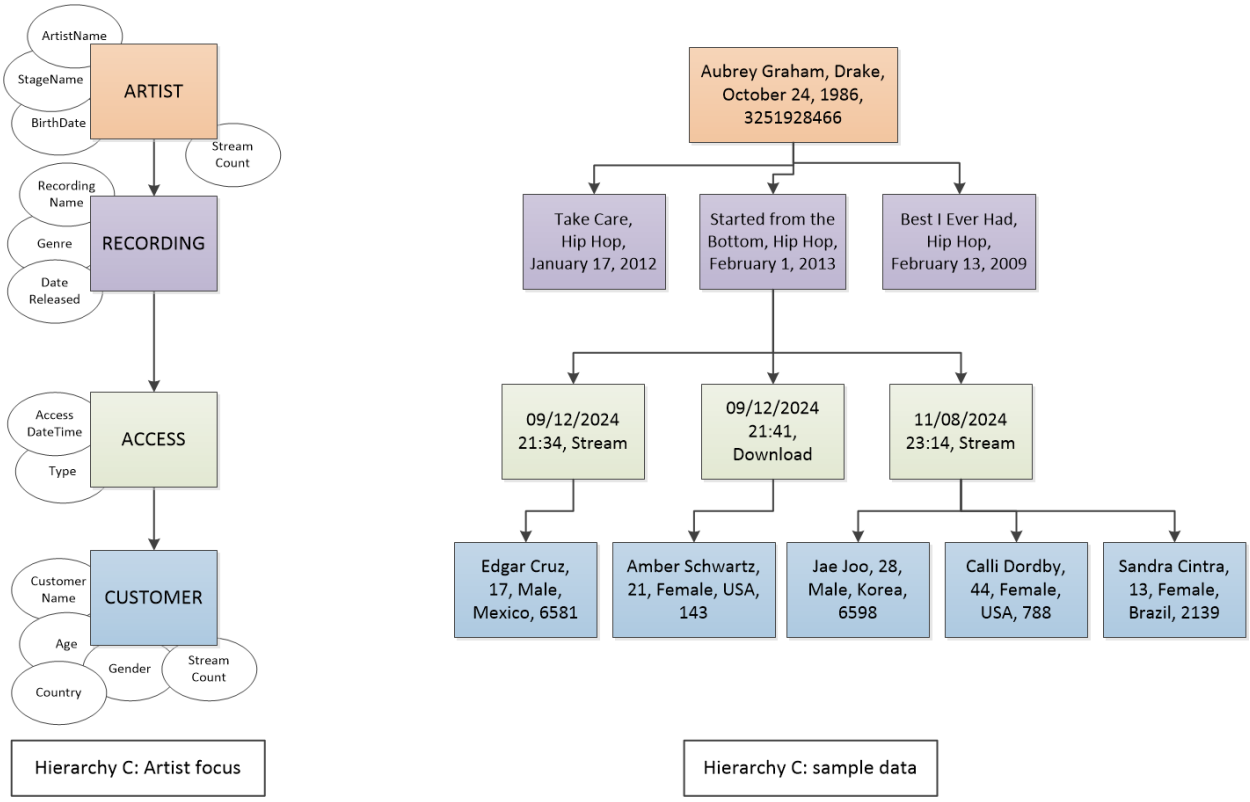

Figure 2.2: Example hierarchy of artist-centric data for MUSIC_STREAM

Remember please that the examples in Figures 2.1 and 2.2 are simple yet convey the incredible challenge people had trying to store data and eventually answer questions! These examples are intended to show how humans had to organize related objects in a top-down manner with a consistent point of reference at the top. In the two examples we see, the entry point of the hierarchy (Neighborhood for the METRO_TRANSIT hierarchy shown in Figure 2.1 and ARTIST for MUSIC_STREAM in Figure 2.2) determines all the rest of the underlying data.

Relational Models: METRO_TRANSIT and MUSIC_STREAM

Even though we have not yet learned how to construct or even read a relational model, it is important to reveal several modern designs so readers can compare the way things are commonly organized today to what we have just seen with hierarchical models. We will see the characteristics in slow motion through the rest of this chapter, but for now, let us see what finished relational models for METRO_TRANSIT and MUSIC_STREAMING databases might look like:

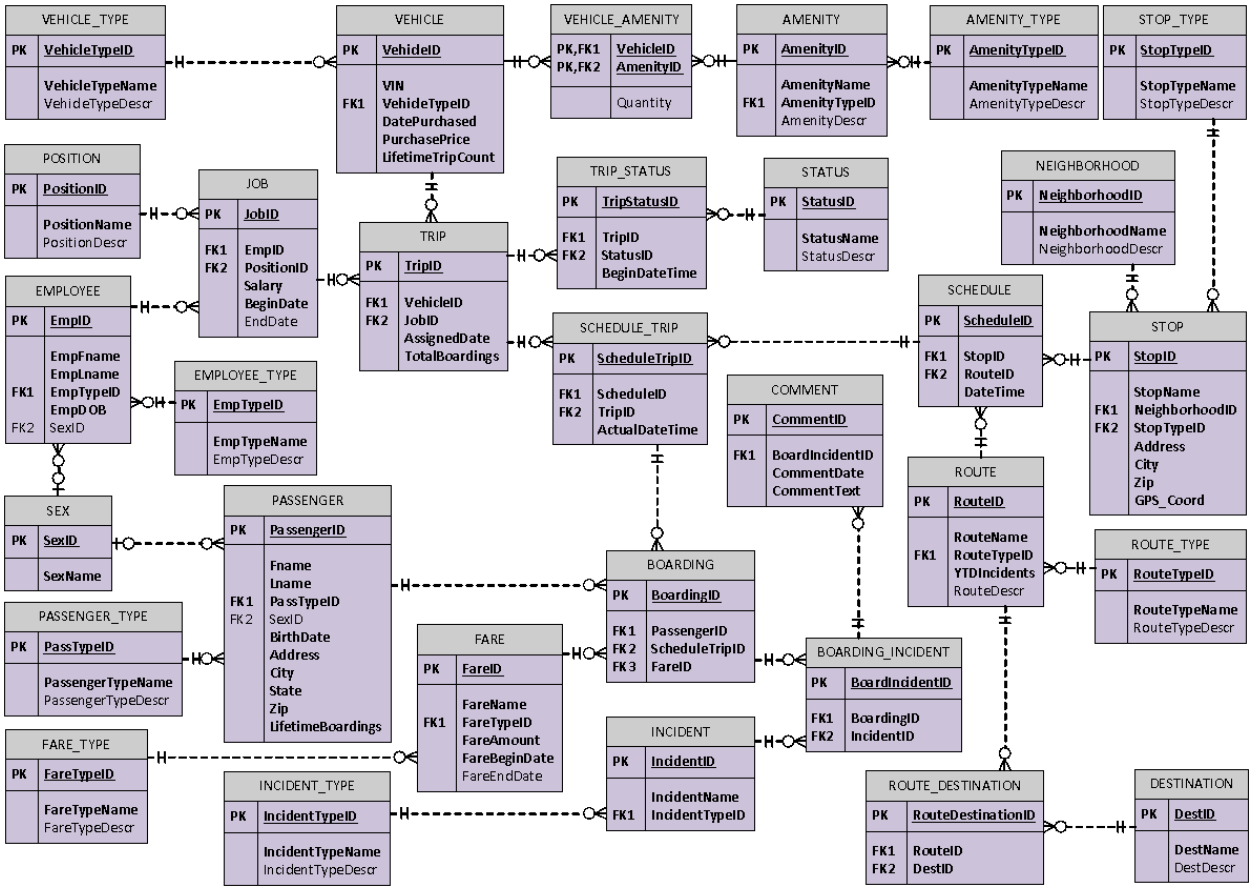

Figure 2.3: Example Relational Data Model of METRO_TRANSIT

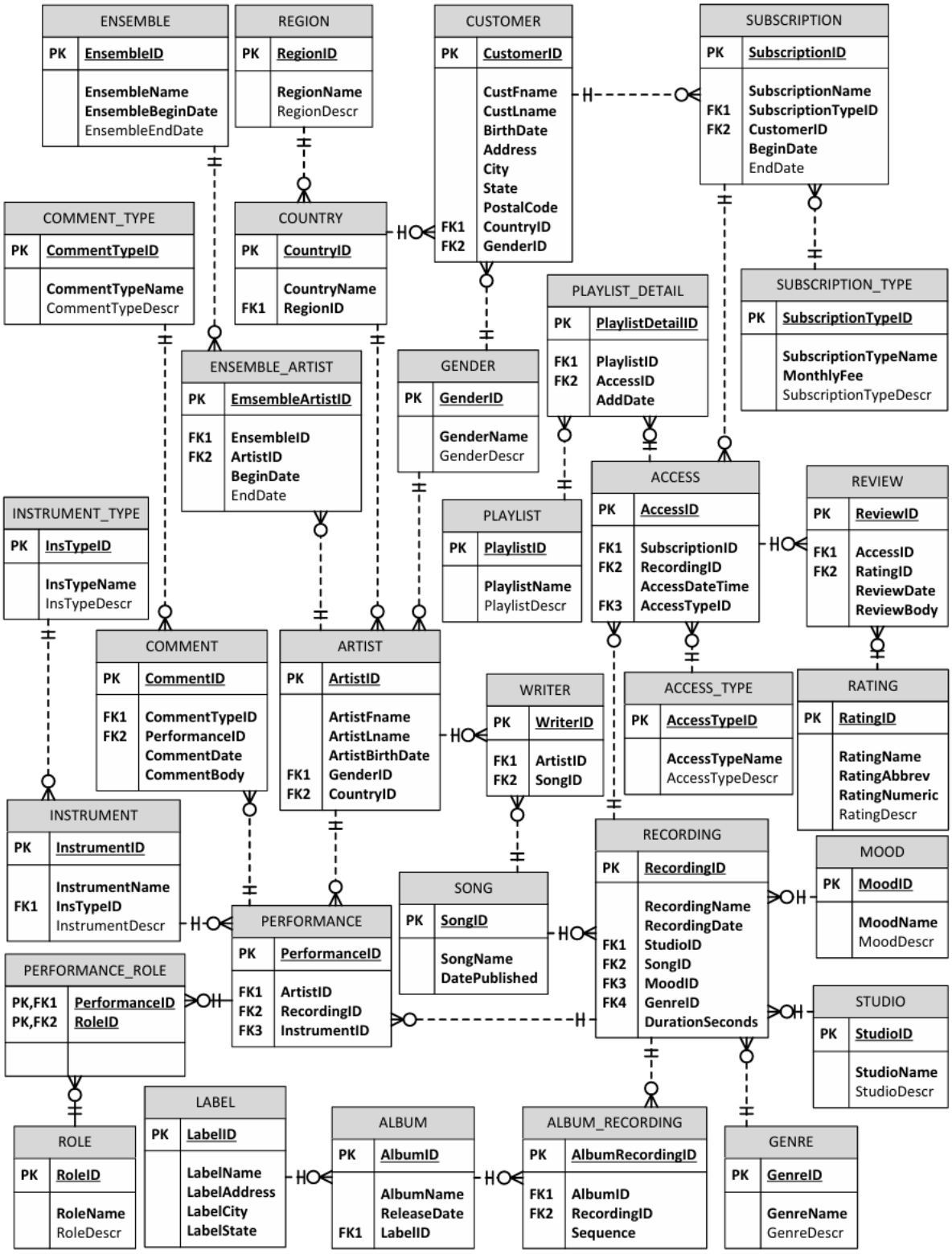

Figure 2.4: Example Relational Data Model of MUSIC_STREAM

Initial explanation

If this is the first time you have looked at an ERD, do not worry; these concepts will be explored in greater depth in the next few pages. The above example in Figure 2.3 for METRO_TRANSIT has 31 'boxes' while next diagram in Figure 2.4 for MUSIC_STREAM has 30. These boxes are referred to as 'entities' and will eventually be called 'tables' once the design is coded and installed within a database management system, like Oracle, Postgres, or Microsoft SQL Server.

Boxes are Entities

Each of the boxes in the diagrams of Figures 2.3 and 2.4 are called 'entities' at this stage of the design. These will eventually be called tables when the final database is built and will contain the actual data in a manner that might be considered a spreadsheet.

An entity should represent or align with a real-world object that business users require data from. Take a quick minute and review some of the objects included in this design; they should all be familiar if you have spent time on public transportation or have listened to music through an application on your phone.

Lines are relationships

The perforated lines between each entity represent a relationship, which means that the two entities (aka 'boxes' for now) at either end of the lines are somehow related. There will be a common column from one of the objects copied into the other to cement this relationship and provide a way to write code pulling data from both that is precise.

The arrow-shaped markings are called "crow's foot" notation which provide clarity around the type of relationship that exists. More on these aspects and ideas of an ERD will come later in this textbook.

The investigation of the relational model starts with the introduction of the concept of keys, including primary key (PK) and foreign keys (FK). The rule to follow with keys is that every table should have a primary key as a method to uniquely identify each row. A foreign key is a reference of a PK placed on a different table and acts as a link between the two. This ability to reference every column of a different table by simply having a copy of the unique identifier is the brilliance of relational theory. More on these keys coming up!

With the relational model, we are no longer designing data on disk in the shape of reports for people to read; there is now a separation between the storage of data and the reading of data by people. Absent the rigid hierarchical structure that defined how data was stored for centuries (both paper systems as well as early computer systems), the relational model stores each object like VEHICLE, EMPLOYEE, ROUTE or ARTIST, RECORDING, and SUBSCRIPTION as a 'stand-alone' table that can be referenced from any other object.

An entity/table represents a single business object that is essential or critical to the operations of the business. We list out attributes of each object that provide the details necessary to make decisions about that object when running the organization, such as names and birthdays of employees or the exact vehicles that were on specific trips at a time of day. Recognize please that the attributes (soon to be called 'columns') of each entity are focused on the data that relates directly to the business object that bears the name of the entity. If we think that additional data from another object is required, that is a sign that we need to create another entity and bring in a foreign key that provides a connection to all the additional data from a related object.

This design significantly reduces the amount of duplicate data required as well as limits the opportunities for typos and abbreviations (also known as 'bad data') to enter the system.

Primary Key

A primary key (PK) is a column or a combination of columns that when compared to every row in a table is guaranteed to be unique. This is a foundational concept for relational theory. We all have ID values: driver's license, StudentID, EmployeeID, AccountID for phone, electric bill, and possibly passport. Colleges, governments and utilities all need to make sure our account is unique, and our payments are sorted and applied based on the activity we charged. Every purchase we make at any store has a unique ProductID for each item we buy as well as a unique OrderID for the same reason; the business must be able to track which items are being sold as well as when to manage their business efficiently.
We know that there must be a unique identifier for the relational design to work well. Each of the tables from the TRANSIT example above have surrogate PKs (even the 'bad design' examples). Technically, a PK does not *have* to be a standalone surrogate key whose sole purpose is to be unique. Believe it not, the primary key just needs to provide uniqueness for each row; this means PK can be any combination of columns that are part of the table already, such as first name, last name, and zip code all together.

While academic argument exists on which is preferable from an elegance and beauty perspective, it is strongly suggested to design all tables with surrogate keys whenever possible. The reasons are that a surrogate key has several performance benefits not available with a composite primary key constructed from multiple columns.

1. The integer data type is narrow (only 4 bytes) which means we can fit more rows in memory than if the combined key of several columns of names (which could easily be ten times wider on average). Basic math tells us that the more food in our fridge (memory) the fewer trips we need to make to the store (disk or cloud). The only difference in this metaphor is for databases, the 'store' is not just down the street in our immediate neighborhood; it is the equivalent of being 200 times further away and about 2 days drive as opposed to a neighborhood store being 9 minutes away!

2. The integer data type can quickly have an 'auto-increment' or IDENTITY feature enabled. This means the assignment of primary key values is automated and no people need to be involved. This speeds up INSERT transactions to the point that millions of rows of data can be processed in an hour. Additionally, the auto-increment is *sequential* (this is secretly very impactful to tables where both READ/WRITE activity are both busy). As you might be aware, WRITE activity (INSERT, UPDATE, and DELETE statements) require exclusive control of the rows affected by the statement logic; if the auto-increment feature is enabled on the PK, then all of the affected rows for the INSERT statement is at the very top of the table, leaving the vast majority of the rest of the rows open for READ activity (SELECT statements…which only need a SHARED lock).

3. The default INDEX on the primary key is a clustered index. This means the data is placed on disk in a physical order of the sequential auto-incremented integer. As we might know, having similar data located physically close together on disk allows for range-searches at the root level of an index (these are like flipping through a dictionary trying to find a particular word and not having to validate the actual words on a page; we trust the index at the top. We can very quickly scan the top-level of the index until we locate the sought-after value. Additional explanations on both data types, locks, and indexes will occur later in this textbook.

Foreign Key

A foreign key (FK) is part of the brilliance of relational theory. As mentioned previously, it allows the referencing of data held in other tables by having a copy of the other table's primary key. We defined correctly the entire contents of the referenced table can be read

when the PK/FK values align. As we have seen, this is much more efficient than copying the related data in other tables throughout the database. Understanding how PK/FK work is the central point of relational theory and by extension also data analytics. It may not be fully digested yet, but do not fret; we will have more examples and practice in later chapters. It is usually not until users start writing queries with JOIN statements that this concept becomes clear.

Assuming we have followed the proper data modeling techniques of normalization (covered in greater depth in chapter 3 *Data Modeling*), we should be comfortable that all appropriate relationships have been determined, and the PK/FK exchange is going in the right direction. While this is perhaps one of the most common mistakes early designers make, even older 'seasoned' engineers like me who have built hundreds of databases can also make a mistake.

Relationships

In a database, tables sometimes have direct relationships where one table contains values that help describe the values in another table. Examples are 'TYPE' tables that help categorize the data in a second table. Consider the ERD earlier in this chapter; transit organizations in large cities may have more than several thousand vehicles in their version of the VEHICLE table (one row for each physical vehicle they operate). The VEHICLE_TYPE table, however, may have fewer than 10 rows! The reason is the VEHICLE_TYPE table simply provides context and clarity as a look-up table. Again, it is intended to prevent typos/abbreviations and allow for very past processing of both reading data to create reports as well as writing new data into the system by avoiding manual data entry of existing categorical data.

Every relationship is established when the PK from one table is placed in a second table as a foreign key. This connects values on a row-by-row basis and effectively extends the reach of one table into the other where the PK and FK columns align.

Cardinality

Cardinality is a label designers use when defining a relationship between entities. We can go further and say cardinality is the definition of the number of occurrences between rows from one table and rows in the other table that participates in the relationship. The most

common relationship definitions will be one-to-many (1:M), one-to-one (1:1), and many-to-many (M:M). Please note that relational theory does not allow M:M relationships to be coded into a finished database! This means that once we identify a relationship as M:M, we have more work to resolve it into multiple 1:M relationships by creating a bridge table in the middle called an 'associative entity'. More on these to come!

Crow's Foot Notation

Please be aware there are two directions or perspectives for each relationship, including 'right-to-left' and 'left-to-right'. Accordingly, there are symbols for each relationship, with some appearing at each endpoint of the line that connects entities. These symbols have an inner and an outer part; one for participation (either mandatory or optional) and the other for multiplicity which is a fancy word for how many occurrences will be on a row-by-row basis for the relationship.

The values for multiplicity will be either '1' (symbolized with a vertical line) or 'many' (symbolized with a crow's foot like a 3-pronged fork). There are only four possible combinations (perhaps eight if thinking in both directions). Let's see some examples clarifying the definitions as well as from within the diagrams we have already been presented with!

Figure 2.5: Example 'Optional Zero to One' relationship read left-to right

Figure 2.6: Example 'Optional Zero to One' relationship read right-to-left

The relationship of 'Optional One' is not very common in a fully normalized database unless a foreign key is determined to not be mandatory or is often unknown. Outside of having a NULL value for a foreign key, having an 'Optional Zero-to-One' relationship indicates an

opportunity to continue exploring whether there might be a 'Many-to-Many' (M:M) relationship between the participating entities over a longer period.

This relationship definition only exists in METRO_TRANSIT regarding the SexID for EMPLOYEE and PASSENGER. In MUSIC_STREAM, there exists a relationship for GenderID for CUSTOMER as well as MoodID in RECORDING (which is subjective). While it could be advantageous to capture this data if it were available, it genuinely may not be available (or relevant) in many instances.

Consider the following definition of a PK-FK relationship:

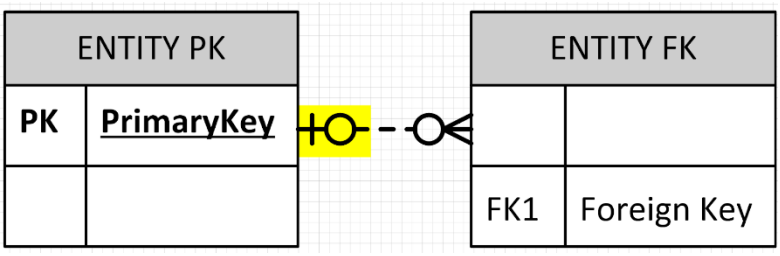

Figure 2.7: Example 'Optional Zero-to-Many' relationship when read left-to-right and 'Optional Zero-to One' when read right-to-left

The relationship depicted in Figure 2.7, the foreign key is 'optional', as indicated in the entity on the right with FK1 in regular font that is not in **bold**. The yellow highlights the changing symbol that shows 'Optional' (the 'O' or zero) versus 'Mandatory' (the vertical line) in Figure 2.8 below.

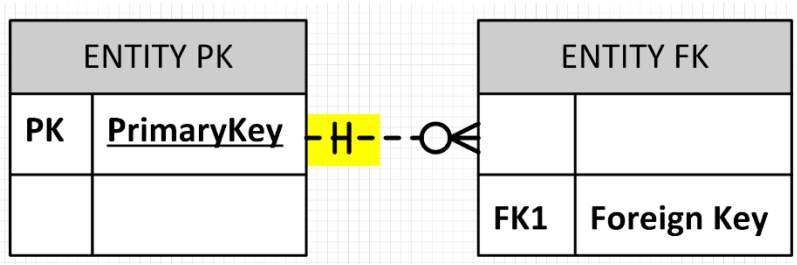

Figure 2.8: Example 'Optional Zero-to-Many' relationship read left-to-right and 'Mandatory One-and-Only One' when read right-to-left

An example where the relationships can be re-interpreted to match an 'Optional Zero-to-One' could be between a BOARDING and INCIDENT (just maybe though). Having this

relationship defined as ‘Optional Zero-to-One’ means that for any single boarding (defined as a passenger getting onto a vehicle, at a specific stop, on a specified trip, with a precise datetime) there can be zero or only a single incident recorded in the database.

While this may make sense at first, it can be argued that one person can have many incidents on a single boarding. Imagine if a passenger tries to evade a fare (incident#1), then gets into a verbal exchange with the driver over incident #1 (making incident #2), then proceeds to fight a second passenger who intervenes incident #2 (creating incident #3 and #4 because a second boarding is now involved) before causing an accident (incident #5)! All this depends of course on how detailed the database needs to be and whether everything just described needs to be tracked as details or just one single incident.

Chances are high if more than 5% of the relationships you define are ‘Optional Zero to One’, you are not evaluating the relationship over a long enough perspective of time. Spend a few more minutes to see if these relationships are better labeled as ‘Optional Zero-to-Many’ when considering a broader range of time (especially if the database will have more than a thousand rows of data).

Figure 2.9: Example ‘Optional Zero to Many’ relationship read right-to-left

Figure 2.10: Example ‘Optional Zero to Many’ relationship read right-to-left

The relationship of ‘Optional Zero-to-Many’ is very common in relational database design and represents the ‘many’ sides of nearly every 1:M relationship. In the examples we have seen with METRO_TRANSIT and MUSIC_STREAM, this relationship is prominent. We will see in most fully normalized designs that most relationships between objects will evaluate to 1:M when accounting for an assessment over many years (instead of a single moment).

We read the relationship in both directions and consider the duration to be the lifetime of the relationship (if not the entire database). For example, in the TRANSIT ERD we can read

the relationship between the entities NEIGHBORHOOD and STOP (seen in the upper right of the ERD in Figure 2.3 with an excerpt included below in Figure 2.11) as follows:

"A NEIGHBORHOOD can have between zero and many STOPs, and the same STOP can be of one and only NEIGHBORHOOD for the lifetime of the database". This is defined as a 'One-to-Many' relationship and written 1:M.

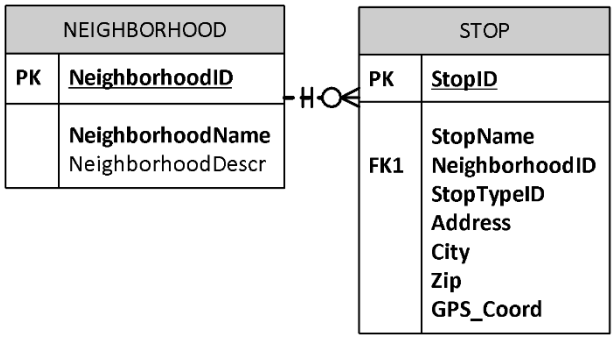

Figure 2.11: Example depicting 1:M relationship

Again, when we articulate the relationship between two entities, we read both left-to-right and right-to-left punctuating the symbol opposite from whichever entity we start with. For example, reading left-to-right will describe the relationship NEIGHBORHOOD has with STOP. This is articulated as "One NEIGHBORHOOD has optionally zero to many STOPs". Please note we look past the two vertical lines immediately next to NEIGHBORHOOD as they describe the relationship in the other direction.
The two vertical lines mean 'Mandatory One' and describe the relationship a STOP has with a row in NEIGHBORHOOD. It is articulated when the relationship is read right-to-left.

Is it possible a city will rename neighborhoods? I suppose it is possible. If that ever happens, we probably will not need to maintain the history of the previous neighborhood name and will just simply update the NeighborhoodName in the entity NEIGHBORHOOD if the previous name is changed or possibly the NeighborhoodID FK value in the entity STOP if the boundaries of an existing neighborhood has expanded.

PK/FK is how all tables are joined together in a multi-table query. The exact syntax will be seen in significant detail in chapter 5, *Beginning with SQL*.

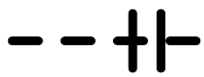

Figure 2.12: Example 'Mandatory One and Only One' relationship read left-to-right

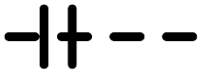

Figure 2.13: Example 'Mandatory One and Only One' relationship read right-to-left

The relationship definition of "Mandatory One and Only One" is also very common in a fully normalized relational database. This represents the primary key side of a relationship that sends a value as a foreign key to the other entity. Whenever we establish a relationship (and are therefore bringing in a foreign key value to one of the entities), there should only ever be one foreign key value for the lifetime of the relationship. This includes an FK value that changes over time!

Looking at the two diagrams we have seen so far in Figure 2.3 METRO_TRANSIT and Figure 2.4 MUSIC_STREAM, there are dozens of relationships defined that have 'Mandatory One and Only One' foreign key value. In my mind, a foreign key value should rarely (if ever) change if it was correctly defined in the first place. An example is correctly an obvious mistake such as erroneously placing the home country of musician Drake as United States; he is from Canada.

If we find ourselves changing a foreign key value more than once due to changes in the underlying data (like STATUS of a single TRIP), we most likely made a mistake in the relationship definition. Frequently updating a value like StatusID, is very impactful to performance due to the need to gain exclusive access to the data (I will compare updates to 'changing a car tire on the freeway' later in this book). We try to avoid forced updates whenever possible.

Using Status as an example, if we mistakenly (in my interpretation) defined the relationship between TRIP and STATUS 1:M as opposed to M:M, it would look as follows:

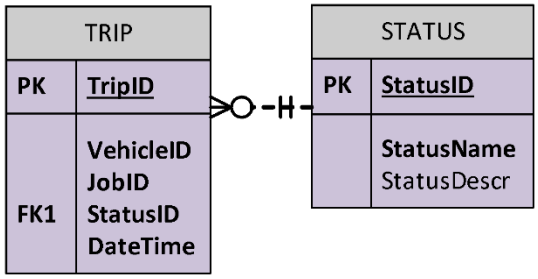

Figure 2.14: Mistake in defining TRIP and STATUS relationship as 1:M instead of M:M

When we define the relationship between TRIP and STATUS as 1:M, we are overwriting the StatusID for a TRIP each time the value changes. We are effectively destroying the previous record of its previous status. You may argue the previous value for StatusID is no longer relevant since no one should care if the bus WAS late if it is now on time, right? When we destroy data, we lose the opportunity to analyze! Think about what we can investigate: the frequency or duration of each different status, potential causes, patterns of occurrence and other critical components. Any data that was important enough to capture in the first place, should be held for analytical purposes later.

Please note that one single TRIP with 12 scheduled STOPs may in fact be on time for most stops but either early or late for a handful of others. A better design interpretation (again, in my humble opinion) is to define the relationship as M:M and capture the exact time the status changed. With this design choice, we will eliminate the invasive updates that require exclusive locks as well as preserve previous values written which allow analyses on the root cause of why the status changed.

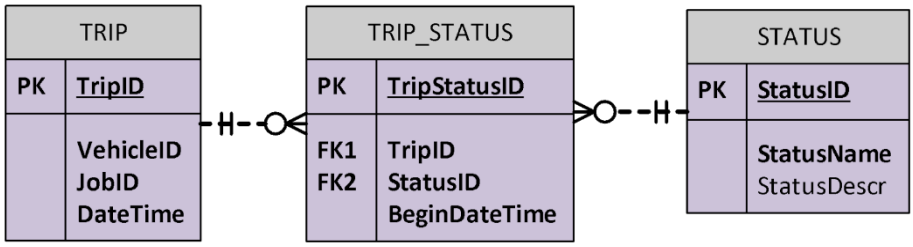

Figure 2.15: Correct definition of TRIP and STATUS relationship as M:M with TRIP_STATUS included as resolving associative entity

Please note that the true relationship between TRIP and STATUS is M:M; one TRIP can have many statuses over its lifetime, and the same STATUS (such as on-track, early, or late) can be held by any number of trips. We will see in chapter 3 *Data Modeling* how to resolve M:M relationships into two that are 1:M by creating a middle table called an associative entity.

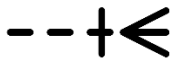

Figure 2.16: Example 'Mandatory Many' relationship read left-to-right

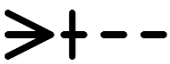

Figure 2.17: Example 'Mandatory Many' relationship read right-to-left

The relationship definition of 'Mandatory Many' is relatively uncommon in relational database design. In the two diagrams presented so far (again, METRO_TRANSIT and MUSIC_STREAM), there are no occurrences of this definition in any of the relationships.

While it may seem that there should be frequent use of this definition, it is reserved for complex relationships where we MUST have values from all participating entities for a record to be captured. For example, let us imagine METRO_TRANSIT with a requirement that each boarding is for 2 or more people. This requires a new entity called RESERVATION added to the diagram. See Figure 2.18 below:

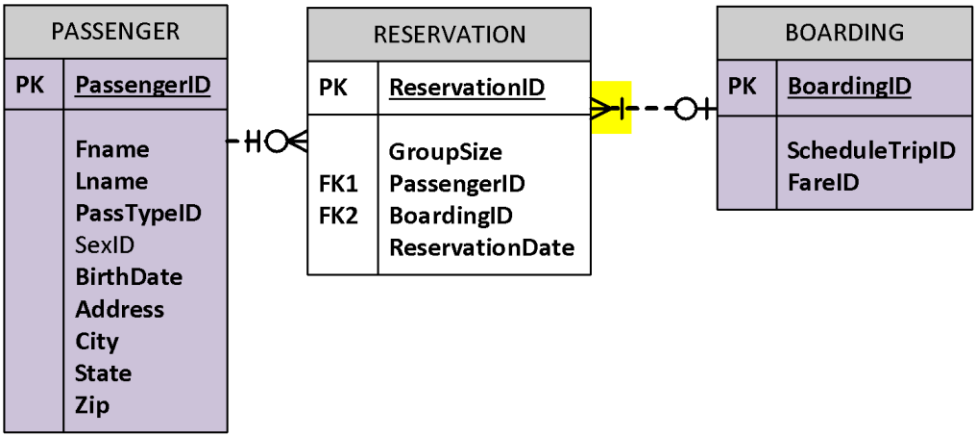

Figure 2.18: An example relationship of 'Mandatory Many' between BOARDING and PASSENGER by adding new entity RESERVATION

This example shows that all rows in the BOARDING table may only exist in conjunction with a row in RESERVATION; these would be entered into the database simultaneously within the same transaction. Since this is just an example to illustrate the 'Mandatory Many' relationship definition, please ignore the presence of RESERVATION as an entity in METRO_TRANSIT moving forward.

Data Types

Somewhere between understanding the basics of the relational data model and constructing one on your own is having clarity on what makes up data types. As it has been decided to include this brief overview here as opposed to stacking too much into other chapters, this section is intended to get every reader prepared for the forthcoming chapters on data modeling and writing SQL code. This overview is relatively brief as it is not intended to be a definitive resource on each of the dozens of data types available. Please continue researching online if there are questions unanswered here.

Anytime a relational database is being considered for development, all stakeholders must understand the purpose, intent, and primary uses the data will provide. I ask all my students to begin their design quests by answering two questions:

- Who is going to use this database?
- What decisions are they going to make from the data?

We will see in chapter 3 *Data Modeling and Normalization* a design process called normalization, but before we get there, we need to have a basic understanding of data types. These are the definitions of the 'shape' of data for each column of a table. We will also see that it is not enough to just come up with a list of objects and call them entities; we must know ahead of time the elements, characteristics, or attributes of each entity as we are doing the design. This requires that engineers know ahead of time how the data will be read or included in queries that will provide data for dashboards and reports fed to key decision-makers.

Data Types are one of the most critical decisions engineers make when designing a new database. Not only can they prevent any number of unintentional human errors, but data types also keep the collection of data in each table structured and better able to be used in an intentional manner when it comes time to write queries and display reports.

Across all relational database products, data types provide the first layer of security against 'bad data' by establishing a pre-set shape for all values in a single column. Along with establishing maximum length of characters and range of permissible values, we can loosely say a data type supports the 'domain' of a column. Having control at the domain level can prevent having 'January' as a phone number for example or a 300-word paragraph as someone's last name.

Beyond the obvious 'clean data' restrictions everyone desires, there are often nuanced controls that each industry or application designer needs. Examples may include the number of decimals for columns defined as numeric values for engineering or scientific research. Columns defined with currency ('money') data types frequently provide benefits by allowing greater programmatic controls by a larger number of financial applications. Obviously, global exchange rates are volatile, and precise controls are required to avoid rounding or truncation when not intended.

We can even create our own custom data type to match effectively any situation mandated by our organization or industry. An example might be mandatory formats for driver's licenses which often are a set pattern that includes a mixed combination of letters and numbers in a sequence. Many industries have their own patterns and sequences for data values that may appear to be impossible to control through an application but become much easier when defined as a data type.

Categories of Data Types

In life, there are many different types of data beyond the written word. These include obviously audio files, photographs, and other visual artifacts like maps, charts, and diagrams of any kind (I am trying to be overly broad here but probably am missing something like x-rays or QR codes). The point I want to convey is, not only are there many kinds of data, but we need to label them correctly for the purpose of learning quickly.

Relational databases have a 'sweet spot' of tracking data from repetitive events we call transactions. These transactions are very common, from buying anything online, 'badging in' at a locked door at work, texting someone from our phone, or crossing a toll bridge. While many of these transactions may also include corroborating data like a picture, the core of these events are words and numbers.

With this delineation in mind, let's take a quick look at the range of data types most common in relational databases (with perhaps a slight bias for Microsoft SQL Server with which I have invested a large amount of time since 1990). These include the following:

- Numeric (whole, exact, and approximate)
- Date and Time
- String (letters and numbers without intention of 'doing math')
- Other data types

Numeric data types

There are many reasons to capture numbers with each transaction recorded into a relational database. These include the price of a product, the date and time that it was ordered, as well as the cost. Many factors will determine which data type is most appropriate; perhaps the most important being whether a user is ever going to include the value in any math equation. This may seem like a strange condition, until we evaluate all columns that hold numeric values; consider the following intentionally absurd and silly math questions from a classroom of students:

1. What is the average phone number?
2. What is the result of adding three dorm room numbers together?
3. What is the result of multiplying everyone's zip code?
4. What are the total StudentID values for people arranged by eye color?

Obviously, while we may be able to calculate an answer to each of the questions above, none of the results may be useful or allow for any effective learning. One of the validation steps when assigning a data type to any column is determining how often there will be any math applied to its values. Surprisingly, perhaps, very few columns will be involved with math being applied to its values. Looking at the data model for METRO_TRANSIT in Figure 2.3 there are only four columns in the entire database (outside of date, time or computed columns) that will have frequent math.

- FARE.FareAmount
- JOB.Salary
- VEHICLE.PurchasePrice
- VEHICLE_AMENITY.Quantity

In the MUSIC_STREAM model in Figure 2.4, there are also only four columns that are 'math-friendly' that are not computed columns or date/time centric:

- SUBSCRIPTION_TYPE.MonthlyFee
- RATING.RatingNumeric
- RECORDING.DurationSeconds
- ALBUM_RECORDING.Sequence (maybe…just maybe)

An additional four columns each in METRO_TRANSIT and MUSIC_STREAM are computed columns and not able to be edited by users. These are intended to speed up reporting and have a high demand or frequency of calculations by customers or management for a reporting dashboard somewhere. Computed columns, user-defined functions and stored procedures are considered advanced SQL topics and are beyond the scope of this textbook. Please consider researching online these topics as they represent a more advanced understanding of relational database design and application programming.

Review the data models in Figure 2.3 and 2.4 to locate the following columns:

METRO_TRANSIT (Figure 2.3)

- VEHICLE.LifetimeTripCount
- PASSENGER.LifetimeBoardings
- TRIP.TotalBoardings
- ROUTE.YTDIncidents

MUSIC_STREAM in Figure 2.4

- SUBSCRIPTION.LifetimeAccessCount
- RECORDING.TotalAccessCount
- RECORDING.AverageRating
- ARTIST.TotalAccessCount

So, assuming we have identified the columns that have the probability of being included in math like the columns above, which are the most appropriate data types to assign? That really depends on the range of expected values, so let's roll through the choices and see the best use for each.

Whole numbers and Integers

Integer data type might be the most common choice for capturing numbers inside a relational database. Many real-world items are counted in whole numbers such as the number of passengers who board a bus or the number times that same bus has traveled across a bridge for example. No fractions or decimals! All that is left to determine for these

columns are the highest expected volume of occurrences that could be required to store in the database. Here are the choices we have:

TINYINT

TinyINT has a finite range of 256 whole numbers from the value of 0 through 255. Please note this does not include values less than zero! This range also matches some common restrictions with early versions of Excel spreadsheets including the maximum number of columns or allowable characters in a single cell. While these limitations are no longer present in more recent versions of Excel, it may be a factor in the decision-making process as you determine the range of data to be placed in a column defined as TinyINT.

The best uses for TinyINT are for columns that are tracking values that will never exceed 255 (duh!). Examples from either design presented earlier include the primary key values for most of the look-up or type tables that provide foreign key values and the context to other tables. These include the following from the previous examples:

METRO_TRANSIT (Figure 2.3)

- EMPLOYEE_TYPE
- AMENITY_TYPE
- STATUS

MUSIC_SUBSCRIPTION (Figure 2.4)

- ACCESS_TYPE
- SUBSCRIPTION_TYPE
- REGION

Projecting columns that do not yet exist in either database we have seen may include a column that tracks the number of hours per week a driver is scheduled to work, or the number of incidents that occurred in relation to a single BOARDING.

Unfortunately, TinyINT is not very common in practical use; many designers will opt for a more flexible data type like INTEGER to avoid surprises of unforeseen values not fitting into their database years after it was originally designed.

SMALLINT

SmallINT has a finite range of whole numbers from the value of -32,768 through 32,767. Like TinyINT, SmallINT can be used for situations where the values of a column will never reach the tens of thousands. Examples from the database designs we have explored may include TotalBoardings for a single TRIP in METRO_TRANSIT as well as DurationSeconds for a single RECORDING in MUSIC_STREAM.

INTEGER

Integer has a finite range of whole numbers from the value of -2,147,483,648 through 2,147,483,647. This data type may be the most common for use as a primary key and foreign key definition as we can very easily make the values automatically increment in a sequential manner (with no people having to be involved).

INTEGER data type is also a solid choice for many columns that are tracking measurements of cumulative events such as recordings a single person has streamed over the life of their MUSIC_STREAM subscription. Please consider that 2+ billion of anything is a large number; most databases will never have any values that reach this number even after 25 years.

Very few non-scientific measurements will exceed the limits of an integer at 2.147 billion, but they have occurred. For example, there are numerous real-life artists (Taylor Swift, Drake, The Weeknd, Ed Sheeran to name a few) that have individual songs(!) that have exceeded this number for total streams.

BIGINT

BigINT has a finite range of whole numbers from the value of -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807. This is a very large number and is intended to capture extremely large data values like primary key values for authentication into busy social media or email systems. Some of these applications are used by more than a third of the people on earth and can see multiple billions of transactions on a normal day. Most organizations will never exhaust the values in this data type.

Numbers with decimals

Not every column that tracks numbers is correctly represented with a whole number; often we need to track data with greater detail. This may include values that track weight, partial consumption of a household utility (such as a water or energy bill), or anything that might be shared. Some columns will be okay with rounding or truncation of the values they contain while others will not. Some columns will require significant total digits for large numbers (called precision) while others may require many digits beyond the decimal point (called scale). We have several data types that each offer best use, including what are known as 'exact' data types and approximate data types.

Exact data types

Two data types that are exact include NUMERIC() and DECIMAL(). The best use for exact data types is when a column of values has a set pattern such as grades for high school and college students as well as nutrient information (like calories, percentage of daily protein per serving) on packaged food items. Exact data types allow the engineer to specify the total number of digits allowed in a column value as well as the total amount of digit after the decimal point. The first of these two settings that appear inside the parentheses are precision and the second is scale.

The best use of these data types is when there will be calculations (think 'math') that will be applied to the columns or values. This will include determining average or cumulative values for weight, dimensions, percentages, duration of expense and similar data. In these cases, accuracy is considered critical.

NUMERIC() and DECIMAL() are essentially interchangeable in use; both allow for flexibility in designing a column that controls the range of acceptable values up to 19 digits combined between precision and scale settings. One slight difference I have read about these exact data types is NUMERIC data type is rigid in the enforcement of its definition while DECIMAL data type will allow values that exceed the limitations of how it was defined. Frequently, designers will substitute one of the exact data types for columns that are tracking money or whole numbers that may be better defined as either INTEGER or one of the money data types.

Again, a slight reminder that this entire section on data types is intended as an overview to establish a basic understanding for all readers heading into the heavier design and coding chapters of this textbook in the next few chapters. Please augment your understanding (if desired) with additional research online. Before exploring the various data types for money and currency, let us take a quick review of approximate data types for numbers.

Approximate data types

Within SQL, there are two different data types that are considered approximate, including FLOAT() and Real. These are often used in scientific calculations when significant digits are greater than 2 (or vary from measurement to measurement) and where the decimal point is not always in the same position.

Summarized differences between FLOAT and REAL are as follows:

- FLOAT has a broader range of values when compared to REAL
- FLOAT is a 64-bit number while REAL is 32-bit (the range of possible values is not 'twice as much' by the way, the bit difference is just the space of memory occupied)
- REAL data type is 'single-precision' while FLOAT is 'double-precision'

- Each float value is also a real number
- Not every real number is able to be stored with a data type of FLOAT

In the databases we have been presented with METRO_TRANSIT and MUSIC_STREAM, there are currently no columns in either database that are assigned a FLOAT or REAL data type. The reason is that the other numeric data types (integer and numeric() mostly) suffice for the ranges of each domain. Examples of potential columns that could be included in these database designs that would benefit from a FLOAT would be as follows:

METRO_TRANSIT (Figure 2.3)

- Revolutions per minute for a VEHICLE engine during a specific TRIP
- Average weight per person on a VEHICLE with passengers at each STOP

MUSIC_STREAM (Figure 2.4)

- Sound frequency for an instrument or vocal performance on a recording
- Streaming bandwidth for a recording as it is being accessed

The measurements mentioned above are not included in the database design because they are not critical to answering the most pressing questions for users of the database. If we changed the perspective of users to engineers concerned about infrastructure, performance, maintenance, or operations then perhaps measurements like these would become a higher priority.

Money data types

People have a justifiable attachment to money; we do not extend trust very far (if at all) when it comes to tracking what people or organizations owe us or what we have been charged for products, services and fees. Therefore, data types that identify money or currency to the database system are critical to presenting data efficiently as nearly every database has at least one column that is designed to store financial data. In the two database designs we have seen so far, in METRO_TRANSIT and MUSIC_STREAM, there are the following columns that store data on money:

METRO_TRANSIT (Figure 2.3)

- FARE.FareAmount
- EMPLOYEE.Salary
- VEHICLE.PurchasePrice

MUSIC_STREAM (Figure 2.4)

- SUBSCRIPTION_TYPE.MonthlyFee

Besides providing transparency to users, other benefits of defining columns storing financial data with 'money' data types are being better able to leverage other measurements that track exchange or conversion rates, annual inflation as well as a myriad of taxes.

smallmoney

Smallmoney is a data type with a range of values from negative 214,748.3648 through positive 214,748.3647. While this range appears to be significant, it should be limited for

columns that are in the low double or perhaps triple digits such as individual food items in a grocery store. In the database designs we have seen there are several columns where smallmoney would be appropriate:

METRO_TRANSIT (Figure 2.3)

- FARE.FareAmount

MUSIC_STREAM (Figure 2.4)

- SUBSCRIPTION_TYPE.MonthlyFee (just maybe)

Specifically looking at EMPLOYEE.Salary, many positions could easily fit under $214,000 per year, but obviously not all. How about in the year 2045 after two decades of inflation? Additionally, not all databases are tracking money as USD; if the local currency is in Japanese yen, Korean won, or Kazakhstani tenge, then the range of numbers in smallmoney data type are severely limited.

Likewise, while most subscription types for the MUSIC_STREAM database will be perhaps $20 to $30 USD a month, consider the possibility of having a subscription type of 'Major University' or 'Corporate Partner' that may have 150,000 users. Is it possible that the monthly fee amount might exceed $214,000? How about in 25 years or after some major technology breakthrough yet to be invented? Best practices design choices are to include a data type that will not cause grief in the future just to save a few bytes in memory today.

money

The money data type has a range of values from negative 922,337,203,685,477.5808 through positive 922,337,203,685,477.5807. This range is significant and covers a vast range of data needs for tracking money (probably even for many countries). In the database designs we have seen there are several columns where the money data type is appropriate:

METRO_TRANSIT (Figure 2.3)

- EMPLOYEE.Salary
- VEHICLE.PurchasePrice

MUSIC_STREAM (Figure 2.4)

- SUBSCRIPTION_TYPE.MonthlyFee

The range for most columns that are tracking money are a good fit for this data type. This includes different countries as mentioned previously where the exchange rate is multiple hundreds or even thousands to 1 USD or EUR.

Date and Time data types

There are many times in nearly every relational database where the transaction details being captured require the context of time to provide the most information. Occasionally, the context just needs to be a simple date such as in the date that a person was born, married, got hired for a job or purchased a house. Sometimes, there needs to be more detail, and the time of day becomes important, such as when an online purchase was made or when a tolled bridge was crossed.

Most relational database management systems have a collection of data types to address the context that individual columns need to provide the most appropriate context across cultures and the different formats that dates and times are presented. Most importantly, we need to let the system know which columns are holding date and time values as we can change the format on the fly as needed.

date

When just the date is required for a column, the data type Date has sufficient range to capture most values ever needed. The data type DATE has a range of values from January 1 from the year '0001' through the year '9999' ending on December 31 with a default format of yyyy-MM-dd. An example is 1942-11-27 (date of birth for Jimi Hendrix!).

time

When the time of an event is best stored separately from the date, there is the data type 'time'. This also has a very precise measurement down to a nano-second with a format of

of HH:mm:ss[.nnnnnnn]. The range of values for the time data type is 00:00:00.0000000 through 23:59:59.9999999. In my experience, the need to measure seven digits deep on each second has infrequent commercial business scenarios. I can see the need for flexibility if there are billions of transactions and a sorted order of sequence is desired as well as for many scientific uses.

smalldatetime

Smalldatetime captures both simple date and time data with a range from the years 1900 to 2079. It has a detailed measurement of hour and minute (no seconds). Quick analysis is this data type could suffice for most columns in almost all databases in the world. Columns that include time values where 'down to the minute' is acceptable are in toll billings, point-of-sale retail purchases as well as badging-in at work.

The biggest potential problem is the granularity of values being shared across many rows of data (potentially thousands of records). With the lack of smaller, more detailed measurements, it will become increasingly difficult to determine the order or sequence of events with this data type in high-volume tables.

datetime

When both date and time are desired for a transaction, there is a combination data type called datetime. This stores a range of values from the year 1753, through 9999. The granularity is as small as three-hundredths of a second (3.33 milliseconds). The format is presented as yyyy-MM-dd HH:mm:ss[.nnn]. An example value is 2025-09-28 17:24:01.110.

In my experience, the datetime format will suffice for greater than 90% of date and time requirements and is effectively a default value in many data modeling and database management systems.

In the database diagrams presented so far, there are several columns of date or time values as seen below:

METRO_TRANSIT: (columns that need just date)

- VEHICLE.PurchasedDate

- JOB.BeginDate
- JOB.EndDate
- COMMENT.CommentDate
- TRIP.AssignedDate
- EMPLOYEE.EmpDOB
- PASSENGER.BirthDate
- FARE.FareBeginDate
- FARE.FareEndDate

METRO_TRANSIT: (columns that need both date and time combined)

- TRIP_STATUS.BeginDateTime
- SCHEDULE.DateTime
- SCHEDULE_TRIP.ActualDateTime

MUSIC_STREAM has 15 columns that track dates (from birthdates to when certain events occurred, like recording date and release date) and only one that requires the specific time (ACCESS.AccessDateTime). In my experience, this 15:1 ratio of dates-only to the need to know the time of an event is accurate.

It is important to know what the potential queries will be when designing a database, especially around dates and times. With a column defined as 'datetime' for example, an analyst can explore what time of the day events are occurring and adjust staffing or other resources accordingly.

datetime2

There are additional data types for the occasional column that holds data with the need for greater granularity of time. Datetime2 provides precise measurements down to 100 nanoseconds while keeping the same date range as 'datetime'.

Datetime2 has the format of yyyy-MM-dd HH:mm:ss[.nnnnnnn]

Datetimeoffset

Occasionally, larger companies are tracking events across the globe and find it important to know the local time in addition to the relationship difference from Greenwich Mean time. The data type 'datetimeoffset' has the same format as datetime2 and includes a measurement to distinguish the local time zone difference from GMT (also known as Coordinated Universal Time/UTC).

Datetimeoffset has a format of yyyy-MM-dd HH:mm:ss[.nnnnnnn] [+|-]HH:mm

timestamp

There is the ability to have a column automatically capture the current date and time whenever a row is inserted or updated. This column often is added to the schema (maximum one per table) to record when a row was last modified. A timestamp datatype captures a unique number generated by the system that gets re-written every time a row is created or updated.

The format for timestamp is YYYY-MM-DD[HH:MM:SS]

Hopefully, you have recognized the importance of being able to work with data and correctly define the context of time often both from a calendar and clock perspective. When we get to chapter 6 *Intermediate SQL*, there will be more detailed discussion regarding retrieving date and time values with date functions.

String Data Types

The most-common data types in my experience are those that store essentially any character found on a computer keyboard. These are known as string data types although sometimes people say 'alpha-numeric'. Since string data types are open to all characters, many inexperienced database designers will misuse string data types, trying to capture all values that may be better assigned a numeric, datetime, or money data type.

When assigned appropriately, string values are often the ones that make sense to people when we are reading data output, as they include names, emails, and phone numbers of customers, the name and address of the store we shopped at plus a list by name of the products we purchased. The string data type even includes the names of the employees that assisted us with our order.

The flexibility of string data types is very attractive as not only can we place any character in them but also up to 8,000 bytes per value. This means that we can store data in Unicode characters which allows many different languages (especially East Asian languages like Chinese, Japanese, and Korean) to be included in the database.

Let's take a quick look at the different string data types available. There are fixed-length and variable-length string data types within the relational model.

Fixed-length and Variable-length

String data types are available in finite or fixed length when the size and shape of the data in a column are the same for each value as well as variable when they are not. Examples for fixed-length values include a 10-character postal code, 2-character state abbreviation, or a 10-digit phone number. These values are known to be an exact length and have the same shape for each row.

When observing the two databases we have seen already, METRO_TRANSIT and MUSIC_STREAM, the only columns that are fixed length are PASSENGER.Zip, STOP.Zip, STOP.GPS_Coord and CUSTOMER.PostalCode. If phone numbers, country codes or perhaps driver license data were being stored, there is a possibility these might also have a fixed length.

Variable-length values are essentially names, addresses, and descriptions for most objects in the database. Looking at the METRO_TRANSIT design in Figures 2.3 as well as the design for MUSIC_STREAM in Figure 2.4, there are more than 40 columns of 'names, addresses, or descriptions' in each database that are defined as variable character (usually as varchar(50) for names and cities and varchar(500) for description columns). Obviously, most names and addresses are inconsistent in length.

There are string data types for non-Unicode data as well as for Unicode. In each case with string data types, the number of bytes to be assigned to each value is placed in the parentheses of the data type definition. The fixed-length Unicode character data type is written with the letter 'n' in front of the data type with the number of bytes assigned to each value up to 8,000 enclosed in the parentheses. The difference between these is that non-Unicode string data type takes a single byte per character stored while Unicode requires 2 bytes per character. Quick math suggests the maximum number of bytes for non-Unicode string data types is 8,000 while Unicode string data types can only go to a maximum of 4,000 as there are two bytes per character.

Fixed-Length

- char(n) non-Unicode data type with a range of 0 -8,000 bytes
- nchar(n) Unicode data type with a range of 0-4,000 characters
- binary(n) non-Unicode data type with range of 1-8,000 bytes

Variable Length

- varchar(n) non-Unicode data type with a range of 0 – 8,000 bytes
- nvarchar(n) Unicode data type with a range of 0 – 4,000 characters
- varbinary(n) non-Unicode data type with range of 1 – 8,000 bytes
- varchar(max) non-Unicode data type with a range of 0 – 2 GB
- nvarchar(max) Unicode data type with a range of 0 – 2 GB
- varbinary(max) non-Unicode data type with a range of 0 – 2 GB

The data commonly assigned binary data type are diagrams, photographic images or audio files such as those with JPEG, PNG, GIF, or mp3 extensions. Data that are encrypted or compressed are also frequently assigned the binary data type.

Other Data Types

Beyond the standard data types that have been presented so far, there are even more available in most relational database management systems. These include XML, cursor, timestamp and even custom-designed data types. These are going to be dependent on whichever database platform you have chosen to use and are a bit outside the scope of this text. If you find that the data types that have been introduced are not able to accommodate

the specific size or shape of the data you are working with, search online under the topics of data types for the product platform you have installed.

Like many things that have flexibility in design, databases have potential conflicts when deciding exact data types. If we make the shape too restrictive, we may inadvertently cause a burden on users having to re-enter data several times. This may unnecessarily cause application performance to suffer and possibly poison the fragile goodwill companies have with their customers or user base. Similarly, if the data type restrictions we select are too broad, we may introduce other problems such as too much unnecessary data creating search and performance issues in addition to raising storage costs.

There is a balance between protecting the data and eventually using the data in queries or dashboard reports. Be kind and considerate to your end-users by trying to keep their work and overhead to a minimum by thinking how the data is going to be ultimately queried. Across an entire enterprise, there are bound to be different uses and fundamental conflicts regarding the shape of data. Do your best, but ultimately, designers must ensure the data is protected and accurate over the ease of use for one single party.

We will see in chapter 6 *Intermediate SQL* that even if we make mistakes at the design level, we can still change data types 'on-the-fly' at the time of writing queries, using the CAST() or CONVERT() functions.

Now that we have been exposed to a broad overview of the relational model, let's take a moment to reflect on the key takeaways for this chapter with a summary of advantages and disadvantages of the structure.

Advantages of Relational Model

There are many advantages and features for using a relational model as a platform for storing data. These include the following:

- The relational model is very good at creating records based on original transactional events in large volumes with millions of new rows per day.

- Repetitive processes like customer orders can be effectively automated with exceptional accuracy.

- The relational model and its vast set of tools are very stable with little variation from the model introduced in 1969.

- The most popular tools for creating and managing relational databases (data modeling, data storage, and reporting) have been built by very large and prominent companies like Oracle, IBM, Amazon, and Microsoft.

- The standard manipulation language (SQL) is relatively easy to learn which allows even non-technical managers the ability to directly engage data.

- The relational model is flexible to be customized to the unique methods of a company as well as to adapt as data needs evolve during an organization's operational lifetime.

Disadvantages of Relational Model

There are several disadvantages when using a relational data model as a platform for storing data. These include the following:

- Engineers need to be aware of and follow the industry standards to get the best results (this requires a baseline of training and experience).

- Users will need basic training to write SQL as it is the de facto lingua franca of relational databases.

- The protections of transactional processing (known as ACID principles) will become an impediment to those developers not savvy or sophisticated in relational design principles. Again, there is a fundamental baseline of basic operational skill required to design and implement a relational database.

Post-Chapter Challenges

Based on your objectives and intentions on learnings from this book, please approach the following challenges as appropriate for your personal objectives.

Track 1 (THINK): Data Tourist Seeking Ancillary Awareness

Please spend a total of 10 minutes reviewing the following questions, exploring your thoughts. These questions and your responses cut to the essence of this chapter:

- How has the relational model helped societies, organizations, and people evolve in terms of learning?

- What aspects of your life would be better served if you had more relevant and accessible information to assist you in decision-making?

- Find examples of the spreadsheets you have designed or work with frequently in both your private and professional life; how are they designed in symmetry with (or against) the principles of the relational model?

- How might you elect to track data in the future with the understanding of the relational model, either with a spreadsheet, text files, or other methods?

Track 2 (WRITE): Dedicated Student or Recent Graduate
Based on your completion of this chapter, WRITE several paragraphs in response to each question; explore these as if you are being asked a similar question during a job interview!

- Which industries (besides technology) were improved with the technological breakthroughs since relational theory was articulated in 1969? How?

- How has the world economy become more robust, resilient, and efficient due to the ability to create (and learn from!) billions of transactions digitally every hour? Provide several specific examples of exact industries and/or companies!

- Reviewing the example database models for METRO_TRANSIT and MUSIC_STREAM in Figures 2.3 and 2.4, what is your reaction to the way the designs store data when compared to how you may have approached capturing similar data prior to reading this chapter?

Track 3 (BUILD): Full Speed Learner Seeking Job
This track targets readers of this book who want to develop professional skills working with data to launch a career or obtain a more satisfying job. Let's continue with the challenge from the last chapter where you were asked to structure a data collection in a spreadsheet to keep track of a large-scale metropolitan transit system.

- Review the different objects in the example relational model for METRO_TRANSIT in Figure 2.3; how is the perspective of the database in the example different from the perspective you took?

- Comparing the objects in your spreadsheet to the example for METRO_TRANSIT, what elements of data are missing (or present) in your spreadsheet?

- Begin to think which one of your hobbies or personal interests might be interesting to explore a bit closer to design an original database. See if you can build an original database along with the reading of this book! Nothing to code or do just yet as we are diving head-first into data modeling in the very next chapter.

Conclusion

In my opinion, history can look back at human evolution and delineate the epochs of 'before relational' and 'post-relational'; the reason is the tremendous breakthrough that the model has provided in terms of creating massive amounts of data that can be retrieved to learn. The speed and volume of data we can capture has exploded by factors in the thousands from when the relational model was first introduced in the late 1960's. I can remember the days before this technology explosion where very few if anyone flew on a jet, had a phone in their pocket, or had ever engaged a computer. Paper dominated every process people did.

The relational model has had a direct impact on each of our lives with every industry impacted! Organizations can create, capture, analyze, and learn at immensely faster speeds! For this reason, very few enterprises find it competitive to be a local shop offering services to people within walking distance. The cost per byte of data is mere fractions from what it used to be thus making computing ('learning') in reach for anyone with a phone and internet access.

Next up is chapter 3 *Data Modeling and Normalization* where readers will explore the process of taking a problem (or 'learning' space) and developing a relational database in a step-by-step manner. This database development includes 'normalization' which might remind many people of eating vegetables as young children; we are told 'it is good for you' but it somehow feels like punishment. I have designed several hundred databases in my career; it is a bit tedious and initially unfriendly, difficult and perhaps intimidating. As few people truly understand how to build relational data structure correctly (many of the same common mistakes are made by very bright people every semester for nearly 20 years!), give it an honest attempt to become at least a well-understood process if not a professional skill you own. It will be quick and painless! Then you can go outside until the streetlights come on, and maybe even get some ice cream before brushing your teeth and heading to bed! 😊