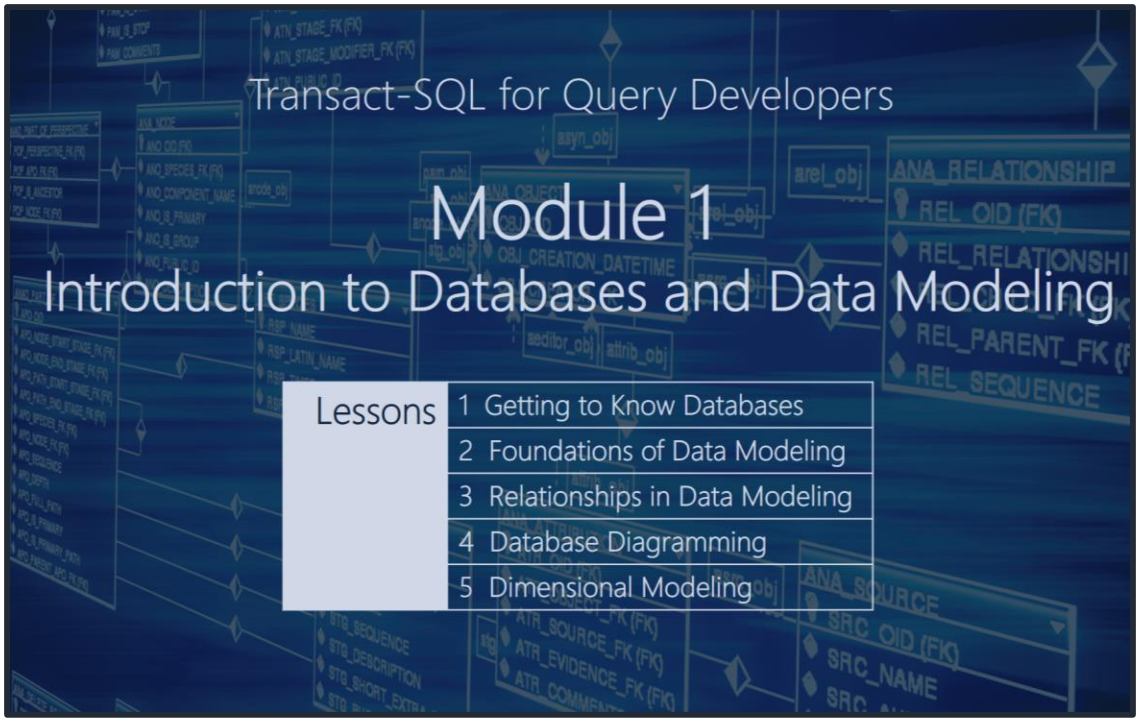
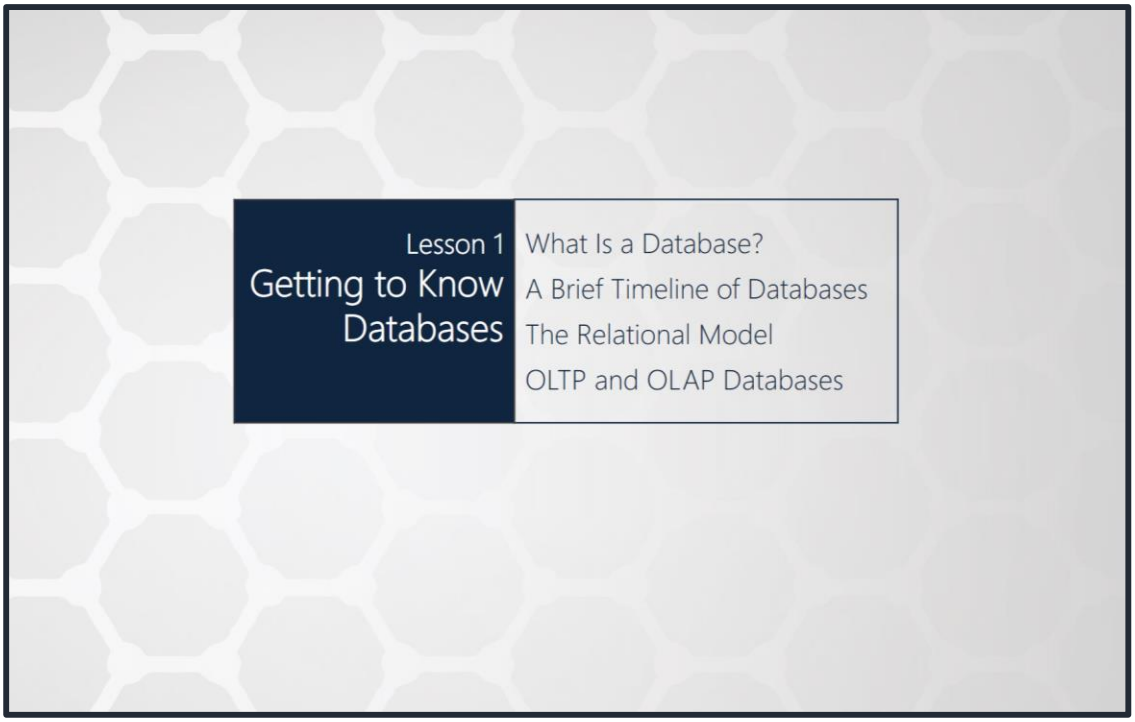


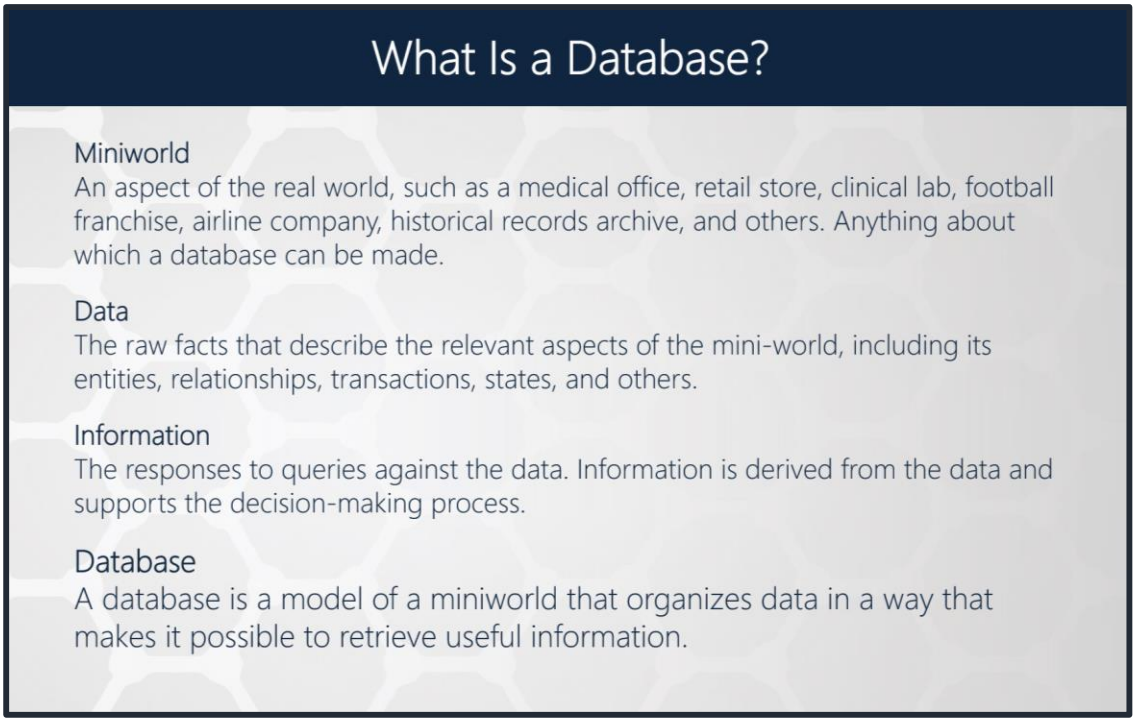
When you're looking for something you need in the garage, it helps if the garage is organized, that it is organized in a way that makes it easy to find what you need, and that you understand what that organization is so that you can navigate through it.

This module will show you how data are organized in a relational database (both OLTP and OLAP) so that you can find information when you need it.



Lesson 1 introduces concepts that will help you understand what databases are and how they are used. A little bit of the history leading to the creation of relational databases will help you appreciate the significant benefits they offer.

Lesson 1 also introduces OLTP and OLAP databases as you will very likely be writing queries in one or both of those types of databases.

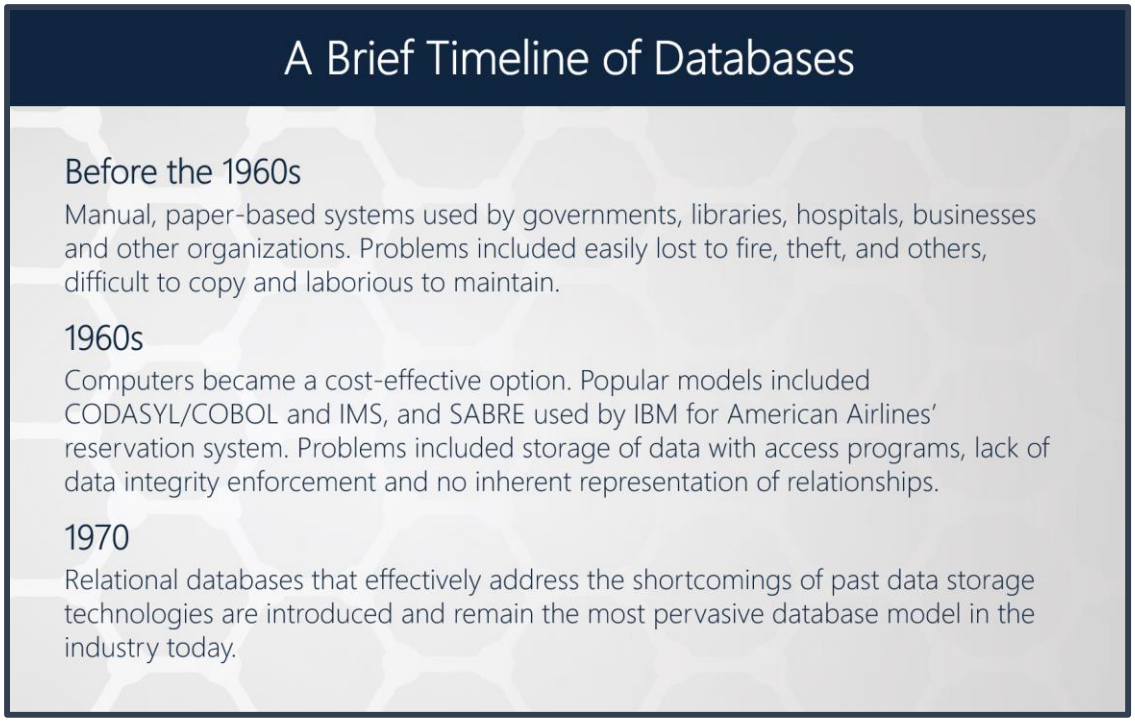


Information is the life blood of any miniworld; it drives the decision-making process. Without information, a miniword cannot function properly.

Information can be obtained by answering questions about the data, called *queries*. Queries can be simple, like, "How many customers do we have?" or they can be more complex, like, "What are the states in which customers placed orders for a product *x* that were shipped via air to California, and in which the shipping date was delayed by more than the average delay of all orders by the same customer? (Do not include products purchased from suppliers who have shipped orders that have arrived on schedule over 85% of the time unless the purchase was to fulfill an order for customer *y*.)"

The answer to both of those questions *informs* decisions—in this case about whether the customer base needs to be increased and whether any action needs to be taken about certain orders being consistently delayed.

A database is essentially an information machine; it derives information from the facts of the miniworld. It does that by acting as a model of a miniworld that organizes data in a way that makes it possible to retrieve information.



Humans have been storing data and deriving information from them for thousands of years. Archeological records tell us that one of our first media was stone tablets. Paper was an improvement that revolutionized our ability to keep records, and up to the 1960s, data storage by governments, libraries, hospitals, businesses, and other organizations was largely manual and in paper form.

Starting in the 1960s, computers became a cost-effective option for faster, more convenient and accurate data storage and retrieval, but the CODASYL/COBOL and IMS systems of that time presented very real obstacles to data integrity. Metadata were stored in programs that accessed the data, so if metadata changed, all data access programs needed to be updated, and if errors were made in any of those programs, data stood a good chance of being lost.

Here is an example of the data section of a COBOL program showing the metadata that defines an ORDER-INFO record. It shows where in a disk file the fields that describe an order can be found.

```
01 – ORDER-INFO.
             03    ORDER-NUM                PIC   9(8).
             03    F-NAME                   PIC   X(25).
             03    L-NAME                   PIC   X(25).
             03    ADDRESS-1                PIC   X(40).
             03    ADDRESS-2                PIC   X(40).
             03    CITY                     PIC   X(40).
```

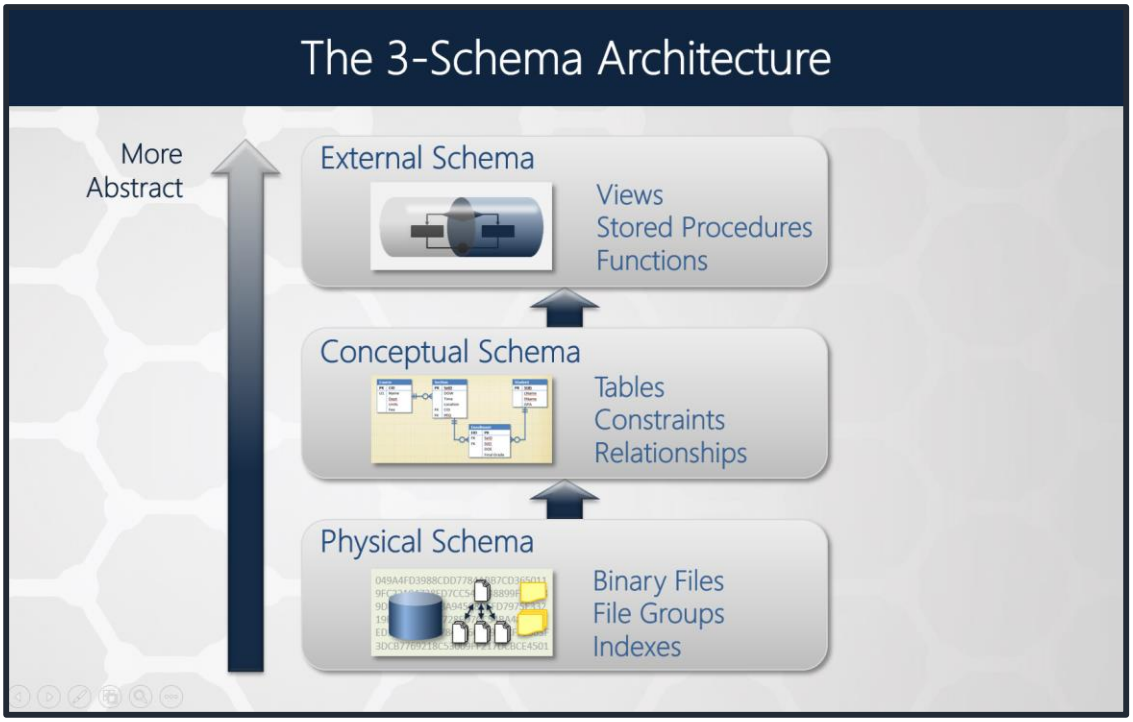
Notice that field sizes are specified, implying the starting position of every field. ORDER-NUM is the first field, so it starts at position 1. It is 8 digits wide (PIC 9(8)), so the next field, F-NAME, will start at position 9, and so on.

Now, imagine that another COBOL program also accesses the same data. It also includes the metadata for the ORDER-INFO record, but it has a typo in the data section that indicates that the ORDER-NUM field is 9 characters long instead of the 8 it's supposed to be. That will move up the starting position of each subsequent field by 1 character. When the program writes data to any of those fields, it will not write it where it should. This can cause the program to overwrite other data and enter incorrect values on updates.

Putting metadata into access programs also creates lots of difficulties for data maintenance. If any changes are made to the structure of the data file, every COBOL program that accesses that data file will need to be updated.

Relational databases were introduced in 1970. Among many other improvements, relational databases centralize metadata in the database. Any access program requesting data identifies the data by name and not by structural aspects of storage. The gatekeeper to the data is the database.

The relational model remains the most successful and popular mode of data storage and information retrieval to this day.



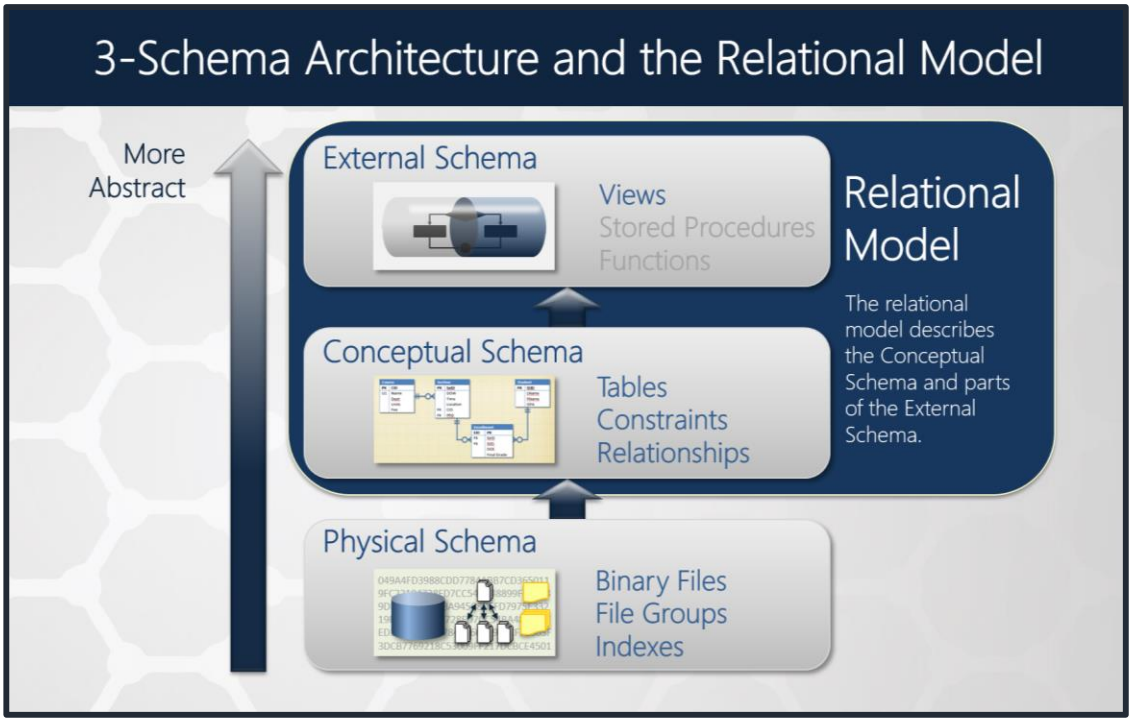
Most relational databases are structured within a 3-schema architecture. Each schema describes the same data but at different levels of abstraction.

At the very lowest end of this "abstraction spectrum" is the physical schema. There is virtually no abstraction there; it is very literal. The physical schema describes the files and file structures that store the database. At the end of the day, the entire database is stored in one or more binary files on the disk. Those files can be organized into file groups, they can be partitioned, compressed, and so on.

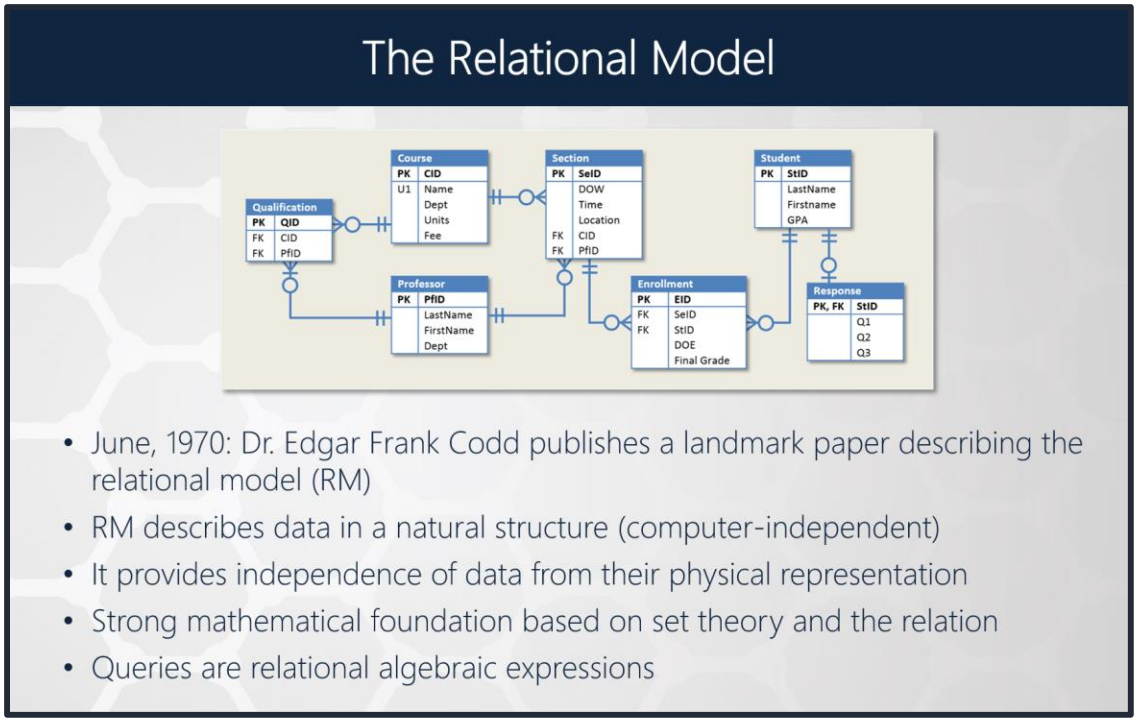
Why do we need anything else? The database is stored as binary data in the files of the physical schema. All of the tables, all of the data in them, the relationships among them, the metadata of the database, all of it is stored in binary format. Working with binary data is not productive. It's very suitable to the computer, but not for humans.

The conceptual schema abstracts the data in the physical schema and makes it easier for us to understand it and work with it. It uses things like tables, columns, rows, primary and foreign keys, constraints, and so on that are easier to work with than binary data is.

The external schema further abstracts the conceptual schema introducing automation and reusability, as well as making it easier for users who do not understand the conceptual schema to work with data. A developer might develop a complex query beyond the grasp of a user, and encapsulate it in a view that the user can query.



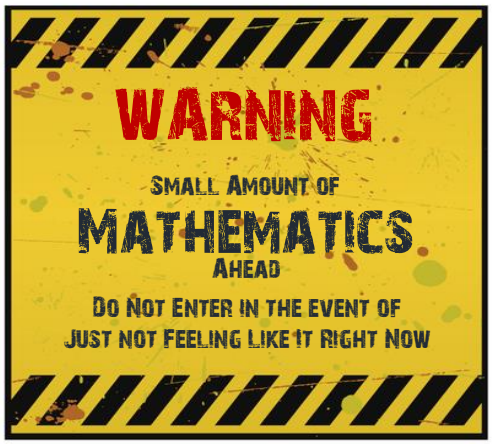
The relational model describes both the Conceptual Schema as well as certain aspects of the External Schema.



In June of 1970, Dr. Edgar Frank Codd, a mathematician working for IBM, published a paper describing a new way of representing and working with data. Dr. Codd's ideas revolutionized the way we work with data, and the relational model he described showed how data could be organized in a self-describing way and on which a variety of operations could be performed. It was a way to abstract the data stored in files on the disk into more natural structures that were easier to manipulate, like tables, columns, rows, relationships, constraints, and others.

The relational model allows us to work with data in conceptual ways that is independent of the way those data are stored in files. This is called *data independence* and is implemeted as a 3-schema architecture that we will soon look at.

The relational model that Dr. Codd developed was described entirely using mathematics. The significance of this is that it allows the relational model to be very flexible in both what it can store and the richness of the information it can produce.



It is not necessary to know a thing about the underlying mathematical structure of the relational model in order to work productively with relational databases and develop effective queries of any complexity. So why bother? Why, indeed! TURN BACK NOW!

There are many good reasons to learn the underlying math. One of them is clarity. For example, take a column in a table. You'll be told in this course that a column represents an attribute of the entity-type that the table represents. And that is a good, sufficient description of an attribute. But look at how an attribute is defined mathematically: An attribute is the name of the role played by a domain in a relation schema. "… the role played by a domain…" That's a very different way of understanding what an attribute is, and also very elucidating.

Understanding the underlying mathematics also helps when developing complex queries. Often, it's best to work through the logic using the mathematical notation without interference from the syntax and other impositions by SQL until the solution is ready to be implemented as a query.

For these reasons and others, what follows is a brief overview of the underlying mathematical concepts of the relational model. Keep in mind that this is just the tip of the iceberg, only enough to give you a taste. For a complete description of the relational model, see Chris Date's book, "An Introduction to Database Systems."

Mathematical Structure

The relational model is based on a structure called the *relation*. Here's the mathematical definition of a relation schema and then a relation:

A relation schema R, denoted as $R(A_1, A_2,..., A_n)$ is a list of attributes in which each attribute A_i is the name of a role played by a doman D_i in the relation schema R.

A relation *r* of the relation schema R, also denoted as r(R), is a set of *n-tuples* $r = \{t_1, t_2,..., t_m\}$. Each n-tuple *t* is an ordered list of values $t = <v_1, v_2,..., v_n>$ in which v_i is a value in D_i. For any *i* and *j* where $1 \le i \le m$ and $1 \le j \le m$, if $i \ne j$ then $ti \ne tj$.

Translation: There's a 2-dimensional object consisting of columns and rows that we call a relation. Each attribute has a domain—the set of values that are allowed in the attribute. Each row (tuple) is a different combination of the values from the column domains.

Databases that implement the relational model are called relational databases. The programs that serve them up and provide a layer of management around them are called relational databse management systems (RDBMS). SQL Server is a RDBMS, Oracle is another one, and so on.

No RDBMS implements the relational model to the letter—the relational model is a methematical structure. Implementations must change things here and there to make the model function in reality. This table shows the equivalence of items among the miniworld, the relational model, and the RDBMS. They're not exactly the same items even though they correspond to each other.

Miniworld Item	Relational Model Item	RDBMS Item
Entity-type	Relation	Table
Domain	Domain	Domain
Attribute	Attribute	Column
Entity	Tuple	Row

Herre's an example: There is a Course table with the following columns: Title, Units, Fee, and Department. The domain of the Title column is any string of characters that is no more than 100 characters wide. The domain of the Units column is 2, 3, or 4. The domain of the Fee column is any dollar amount that is greater than or equal to zero and less than or equal to a cap of $10,000. The domain of the Departmant column is any valid department in the school. Each row in the Course table represents a distinct course; it will have a different combination of the domain values for each column.

Here's a valid instance of that table. Notice that no two rows are alike.

Title	Units	Fee	Dept
Database Management Systems	4	$2,500	CIS
Underwater Photography	2	$1,500	Visal Arts
Accoustic Blues Guitar	3	$2,000	Music

Relationships

Relationships among tuples are represented as relations in which tuples have 2 or more attributes, each one representing a tuple in one of the related relations. The cardinalities of the relationship sets for each tuple define the way relationship attributes are set up in the related relations.

Operations

For operations on relations, Dr. Codd created a *relational algebra*. In addition to the set-theoretic operations of union, intersection, set difference, and Cartesian product, the relational algebra also includes operations designed specifically to work with relations, including the projection (π), selection (σ), join ($\bowtie$), and aggregation ($\mathcal{F}$). All operations on relations produce other relations (an important characteristic of the relational algebra that allows queries to query queries and forms the basis for how subqueries work). Relational agebraic expressions can include single or multiple operations combined in different ways. Rules of commutation and association apply to those operations.

To briefly summarize the relational algebraic operations, the join operation combines two relations into one on join criteria that can be expressed as standard logical expressions using standard relational and logical operators.

The selection operation also uses logical expressions to select or exclude tuples based on their attribute values.

The projection operation then selects the individual attributes from among the combined set (UNION ALL) of attributes in the tuples of the joined relations to include in the resulting relation.

Here are three relational algebraic expressions.

$r_1 \leftarrow$ (Course) (Section)

$r_2 \leftarrow \sigma_{\text{Bldg="B"}} (r_1)$

$r_3 \leftarrow \pi_{\text{Title, Units, Bldg}} (r_2)$

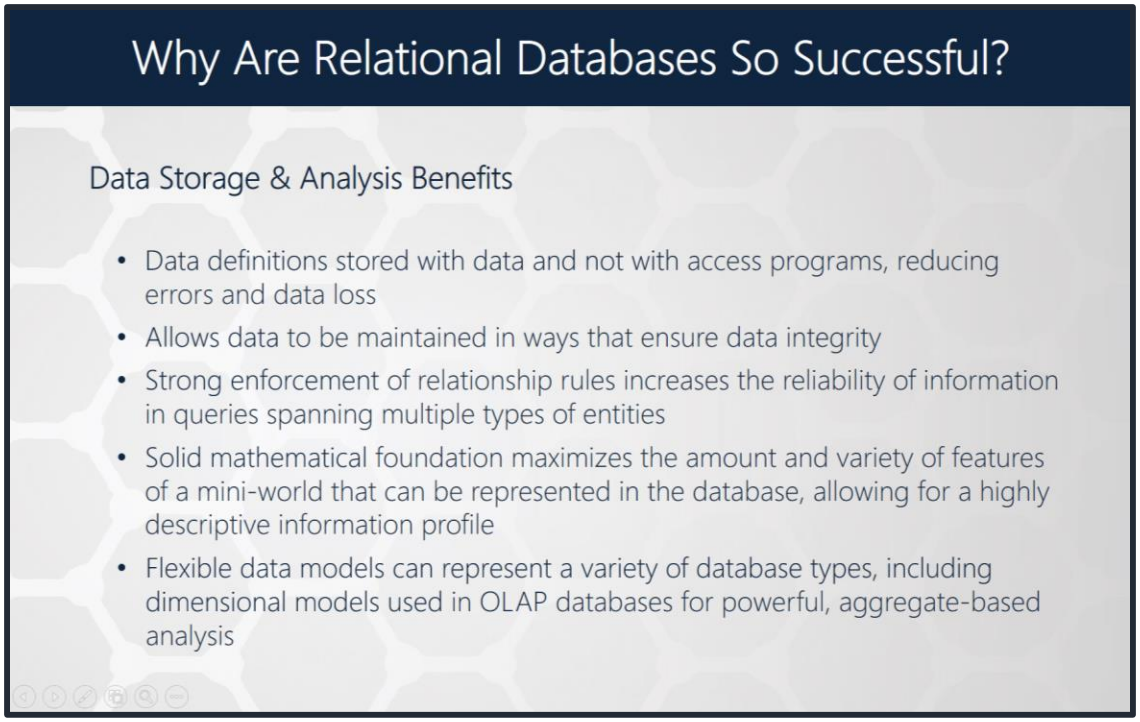
In the first expression, the Course and Section relations are joined using a natural join (on same-named attributes from each relation) resulting in another relation, r_1. The second expression then selects only those tuples in r_1 that have a "B" in their Bldg attribute. In the last expression, the attributes Title, Units, and Bldg are projected into the final result, r_3.

The three expressions can be combined into this single expression:

$r \leftarrow \pi_{\text{Title, Units, Bldg}} (\sigma_{\text{Bldg="B"}} (\text{Course}) \bowtie (\text{Section}))$

SQL implements the relational algebra operations in its SELECT statement. The above expression can be written as an SQL query like this:

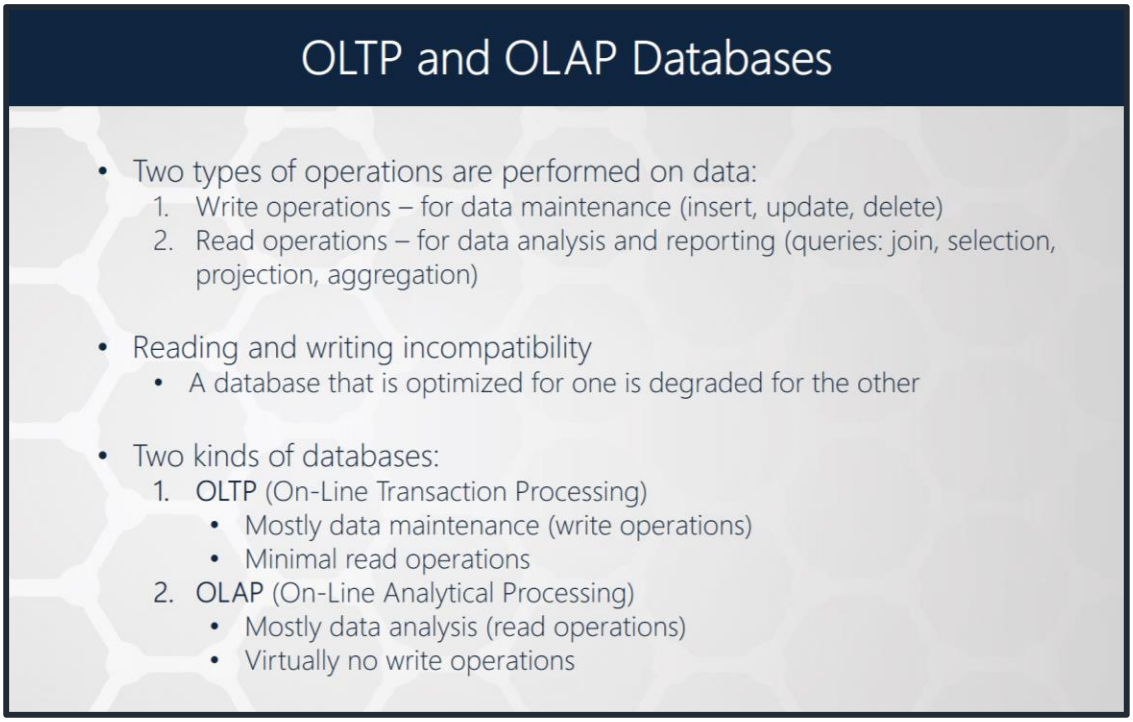
```
SELECT Title, Units, Bldg
FROM Course c
INNER JOIN Section s
    ON s.CourseID = c.CourseID
WHERE Bldg = "B"
```



The centralization of metadata in the database described earlier greatly reduced the potential for errors in the data and assured that all access programs working with that database would play by the same rules.

Unlike the CODASYL/COBOL and IMS systems mentioned earlier, databases are capable of enforcing rules that guarantee data integrity, such as entity integrity (validity of entities), domain integrity (validity of attribute values), and referential integrity (validity of relationships). Strong referential integrity enforcement in particular allows queries to span multiple entity-types, a task requiring some jumping through hoops in the old CODASYL/COBOL and IMS systems.

The mathematical foundation briefly mentioned earlier assures a great deal of flexibility in the relational model, not only to represent a variety of features of a miniworld, but also to work within environments such as OLTP and OLAP that have very different and even opposing needs.



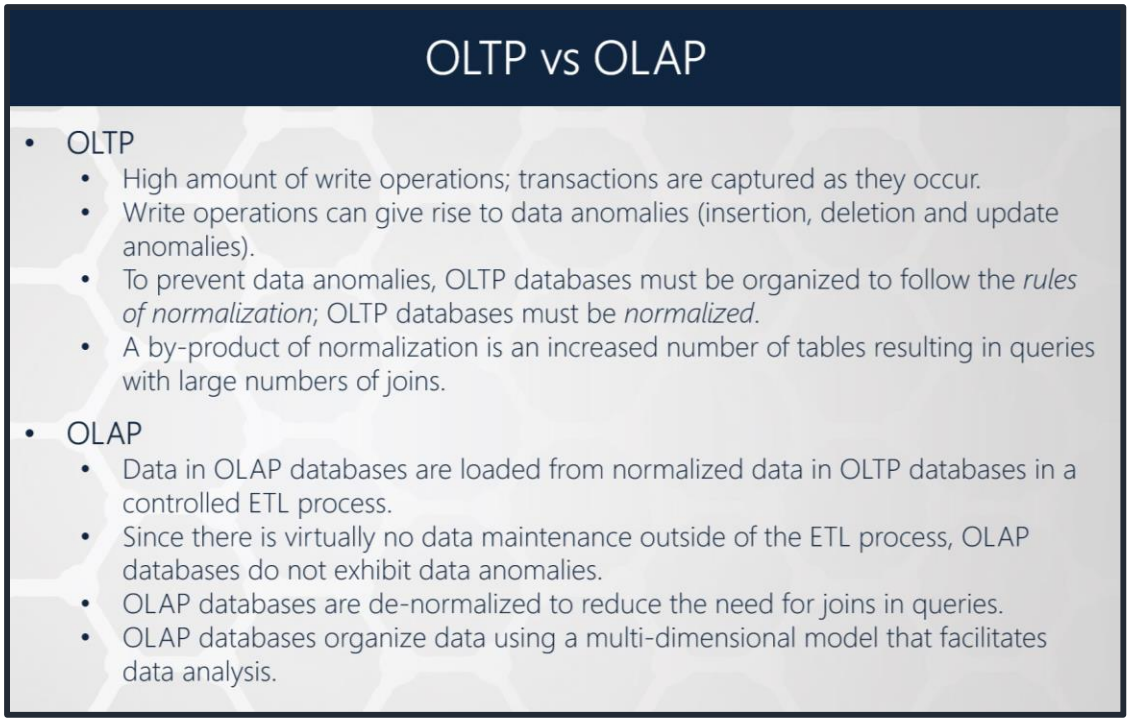
There are two things we do with data: We maintain it and we analyze it to get information.

Data maintenance requires operations that change data in the database. We call those *write* operations and there are three of them—*insert*, *update*, and *delete*. When we create new data, we say that we insert data. When we make changes to existing data we say that we update data; and when we remove data, we say that we delete data.

To analyze data, no write operations are performed, only read operations. Of the two operations, all query operations, including joining data, selecting data, projecting data, and aggregating data require only the ability to read data.

In terms of configuring the database to optimize the operations in the database, reading and writing activities are incompatible—what's good for one is almost always detrimental to the other. Calculated columns are best persisted for read operations, but should not persist for write operations. Queries benefit from materialized views, but then write operations have to maintain multiple copies of the same data.

To deal with this incompatibility, we use two different types of databases. On-Line Transaction Processing (OLTP) databases are set up and optimized for write operations. On-Line Analytical Processing (OLAP) databases are set up and optimized for read operations.



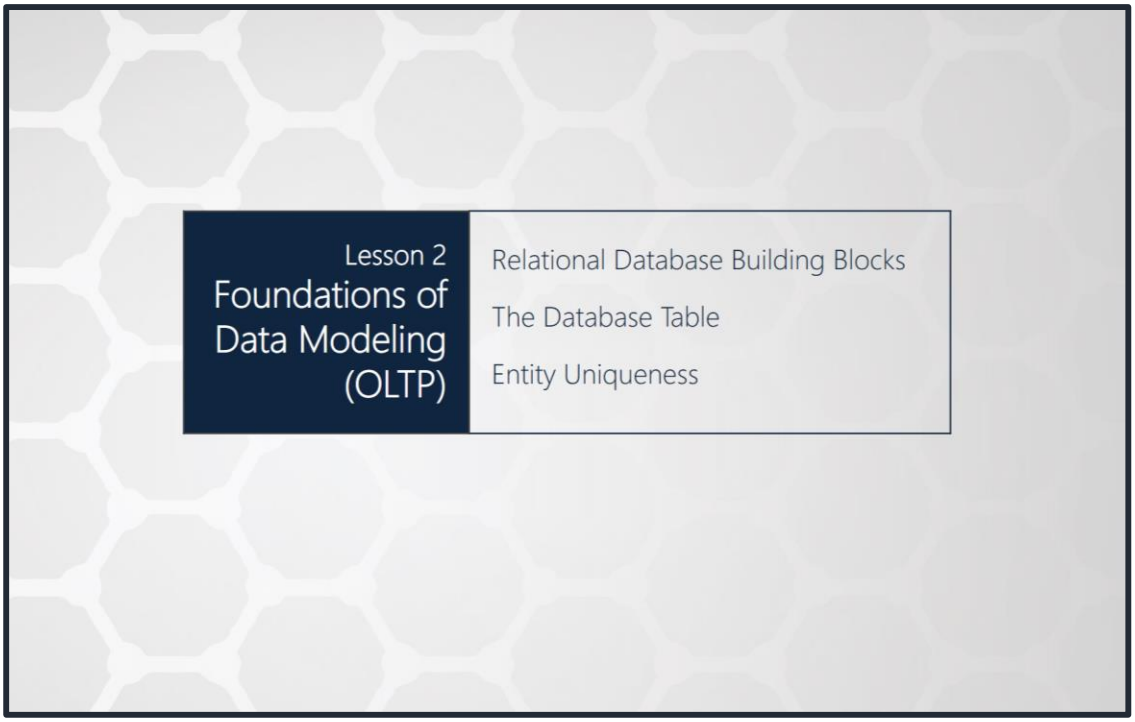
Write operations can give rise to insertion, update, and deletion anomalies, collectively referred to as *data anomalies*. To prevent data anomalies, databases are *normalized*. A normalized database is organized in a way that puts it in compliance with one or more *normal forms*. Each normal form represents a different design rule that prevents anomalies.

Normalization is almost always a requirement in an OLTP database where data maintenance is performed, but it is achieved at a price—a normalized database will have more tables than a *denormalized* database of the same miniworld. More tables require more join operations in queries, and that slows queries down. Again, what works well in an OLTP database gets in the way of read operations performed by queries.

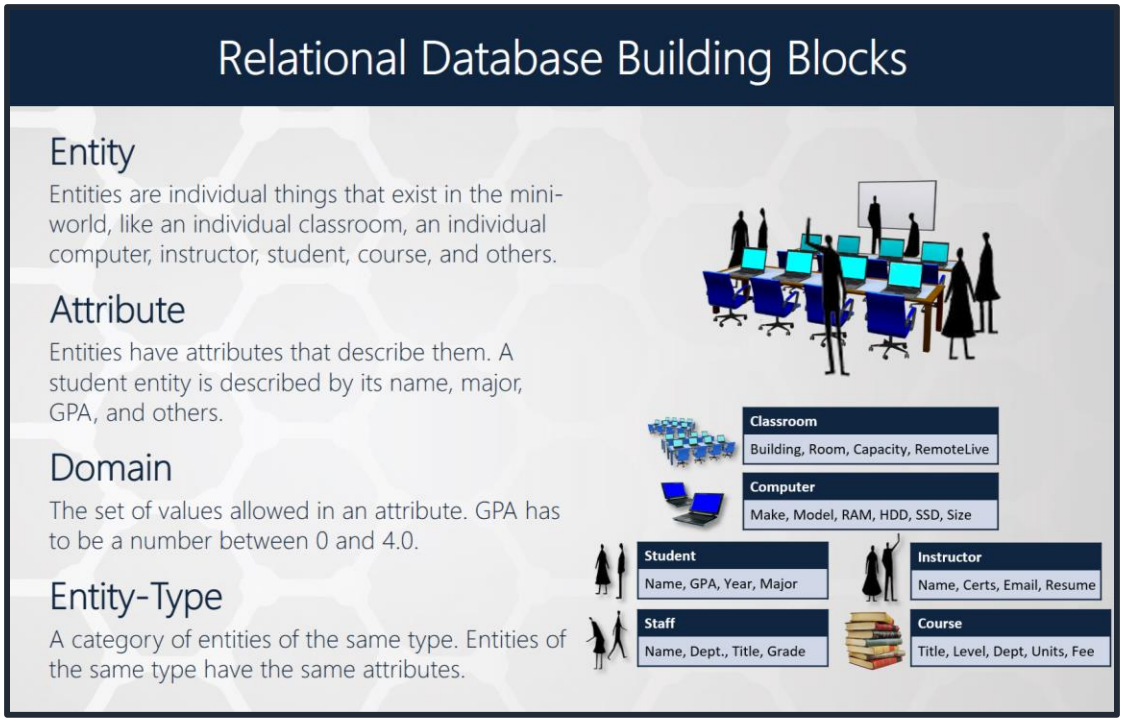
Since normalization is not a requirement for the integrity of data in an OLAP environment where the write operations that can give rise to data anomalies are not being performed, OLAP databases are almost always denormalized. Just like normalization produces more tables, denormalization reduces the number of tables that are needed. That means fewer joins, and simpler and faster queries.

Data integrity in an OLAP environment is assured even without normalization, because OLAP data are loaded through highly controlled ETL (extract, transform and load) processes that extract data from normalized OLTP environments.

At the end of this module, we'll look at the dimensional model; it describes how data are organized in OLAP databases.



This lesson lays down a foundation for understanding how data modeling works. In preparation for the all-important lesson 3, you will learn about the database building blocks—components all databases have that are assembled in different ways to create different databases, a new way of understanding what a database table is, and how to enforce entity uniqueness.



We can say that a house is made of certain *building blocks*—things that all houses have. All houses have doors, walls, windows, kitchens, and so on that, when arranged in different ways, produce different floor plans. It's much the same way with databases. There are building blocks that can be assembled in different ways to produce different databases. Here are the building blocks:

Entity

An entity is something that exists in the mini-world and that has characteristics that are of interest. In a school enrollment mini-world, Prof. Smith is an entity with a name, a department affiliation, a set of credentials, a list of published papers, and other characteristics of interest. In the Interface Technical Training database, you are an individual student entity with a name, contact data, history of courses you may have taken here, areas of interest, and so on. If this were a college, you would also have a GPA characteristic. In an order-processing system, there

Entities don't have to be physical. Courses are also entities, even though you can't see or touch them. Courses also have characteristics of interest, including name, fee, duration, and others. Again, if this were a college setting, courses would have departments they belong to and number of credits they are worth.

If it exists in the mini-world and has characteristics of interest, it is an entity.

Attribute

We've been using the term "characteristic." That's another name for attribute. An attribute is a characteristic of an entity that is of interest.

Attributes must follow two important rules:

1. Attributes must be atomic. Attributes can't be made up of components that have a meaning on their own (when taken out of the attribute). Here's an example:

Attribute	Value
Address	22 Flagg St., Cambridge, MA 02138

Is this atomic? It depends on how the attribute is used. In a miniworld that requires data to be sorted by city, grouped by state, or filtered on zip code, this address would not be atomic because using the data in these ways shows that components of the attribute (city, state and zip code) have meaning on their own.

To fix a non-atomic attribute, an attribute is created for each meaningful component:

Attribute	Value
StreetAddress	22 Flagg St.
City	Cambridge
State	MA
ZipCode	02138

It is far better to assemble things than to take them apart in a database—it is easier, faster, and the outcome is more reliable. Queries that work with components of an attribute have to find them in the attribute value and extract them; that's work that slows the query down.

2. Multivalued attributes are not allowed. Each attribute can have only one value. As an example, the following PhoneNumber attribute violates this rule because it contains more than one phone number:

Attribute	Value(s)
PhoneNumber	555-1234, 555-4321, 555-5678

This rule is important for the same reason as the first rule: It is better from ease-of-use, performance, and reliability standpoints to assemble values than to take them apart.

The fix for a multi-valued attribute will become clear to you after you have learned about one-to-many relationships.

Domain

The domain of an attribute is the set of distinct values allowed in that attribute. A domain might be finite, like the domain of a State attribute, a set of the 50, 2-letter abbreviations for the American states.

Other domains are infinite, with or without constraining boundaries. A GPA domain would be defined as any value between 0 and 4.0, for example, and a LastName domain would have no restrictions on it other than that it must be alphanumeric and allow no more than, let's say, 50 characters.

The relational model defines an attribute as a role played by a domain. Consider a domain of AddressID's (the set of all distinct AddressID's in an Address table). A sales order entity might use that domain for two of its attributes, the ShipToAddress and the BillToAddress. Both attributes draw values from the same domain, but what distinguishes one attribute from the other is the *role* played by the domain. In the first attribute, ShipToAddress, the AddressID domain plays the role of ship-to address, and for the second the same domain plays the role of bill-to address.

Entity-Type

An entity-type is as the name implies—a type of entity. It is defined by its set of attributes.

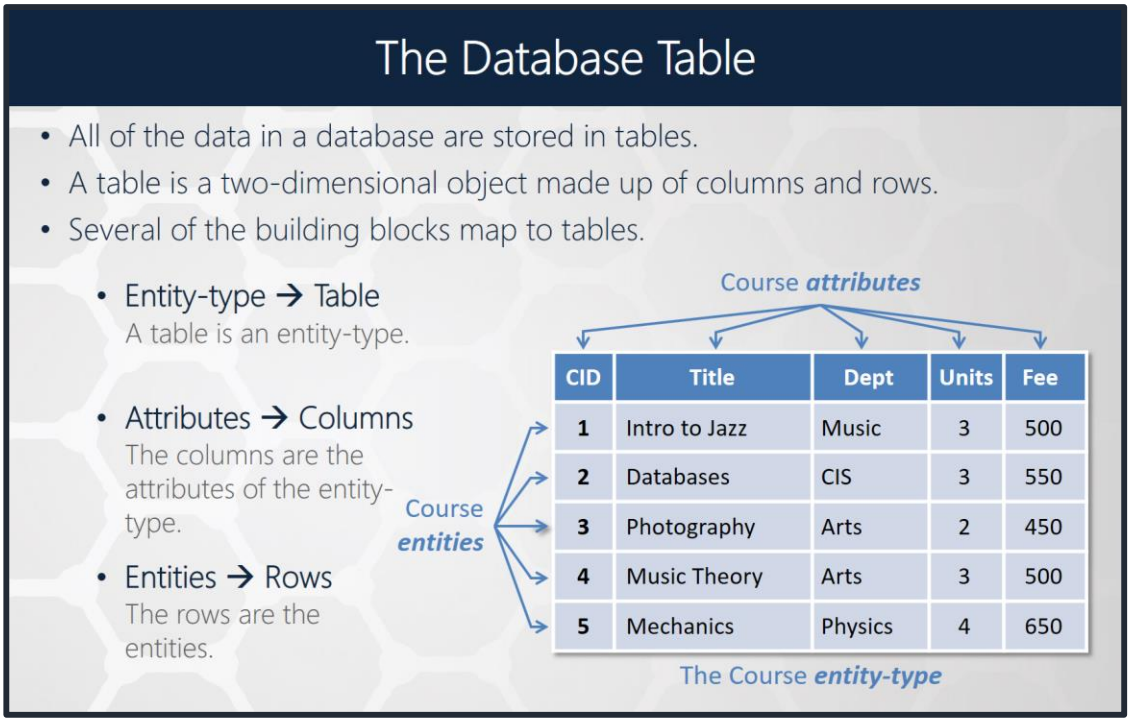
The slide shows a depiction of a classroom. Whatever the course is that is underway in that classroom, it has 5 attributes: CID, name, department, credits, and fee. All course entities have those attributes. We can identify a type of entity that we can call Course; it is defined by the 5 course attributes and each individual course entity is a member of the Course entity-type.

Here are some of the entity-types in a school enrollment mini-world:

Department (Name, Description, DeptHead, Institute)
Course (Name, Department, Credits, Fee)
Professor (LastName, FirstName, Department)
Section (DOW, Time, Location, Course, Professor)
Student (LastName, FirstName, GPA)
Enrollment (Section, Student, DOE, FinalGrade)

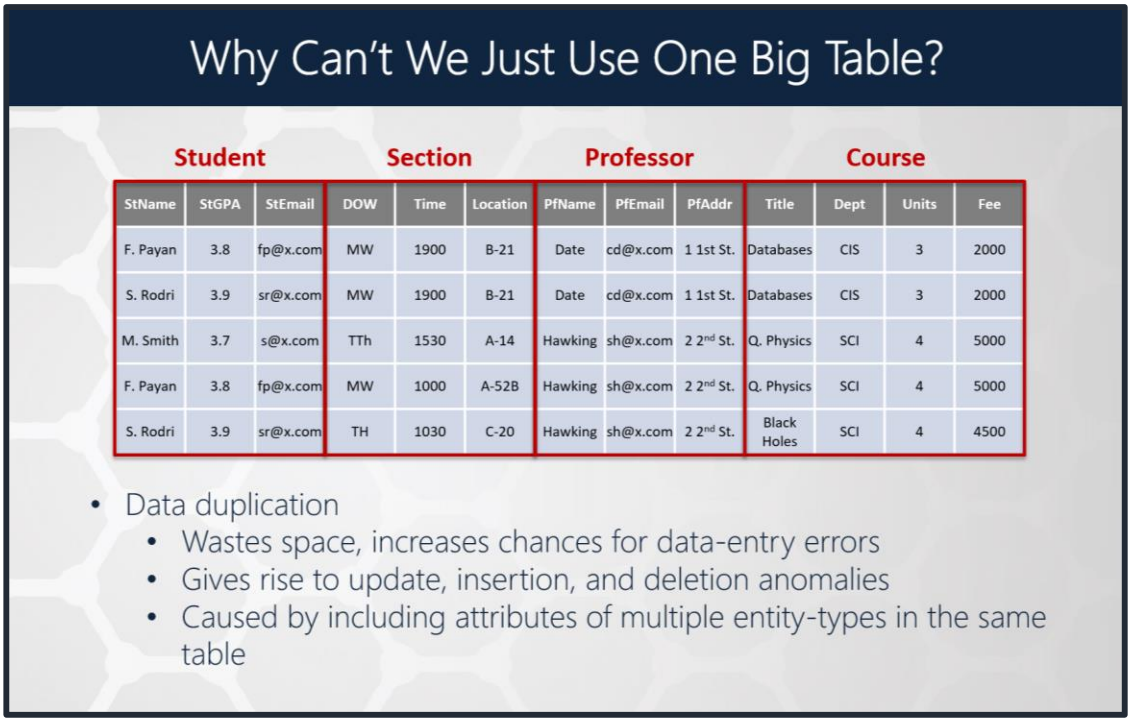
Relationship

Entities interact with each other in different ways. Relationships describe those interactions. Relationships will be explored in great detail later in the next lesson.



Tables represent the entity-types in the miniworld. The columns of a table hold the attributes of the entity-type and the rows hold the individual entities.

Tables are named after the entity-type they implement, so they are properly named in the singular.



A single table implies more than one entity-type in the same table. And that's what leads to problems. Because of relationships among the entity-types, some entities will be repeated in the table. This is actually desirable in an OLAP database as we will see at the end of this module, but it can be fatal in an OLTP database.

In an OLTP database, three data anomalies can occur as a result of combining multiple entity-types into one table, each of which erodes the credibility of the information we get from the data:

Update Anomaly

Let's say F. Payan's email address changes and we need to update the database to reflect that change. If we change F. Payan's email address in row 1, we will create an inconsistency with row 4. We must make the change in more than one row to avoid inconsistencies.

An update anomaly occurs when a change made to a row in a table causes inconsistencies with other rows in the same table. Another way to state the update anomaly is to say that, in order to change a single datum in a table, more than one row in the table must be changed to avoid inconsistencies.

Insertion Anomaly

Let's say that someone walks into our school and asks if we offer any courses on advanced scuba diving in overhead environments. If the table shown in the slide were the entire database, the reply from us would have to be that we don't know if we offer any courses like that because no one has enrolled in any! The absurdity of that lies in the enforcement of an existence dependency (the existence of a course depends on the existence of an enrollment) when that dependency does not exist in the miniworld. It is that absurdity that is the anomaly.

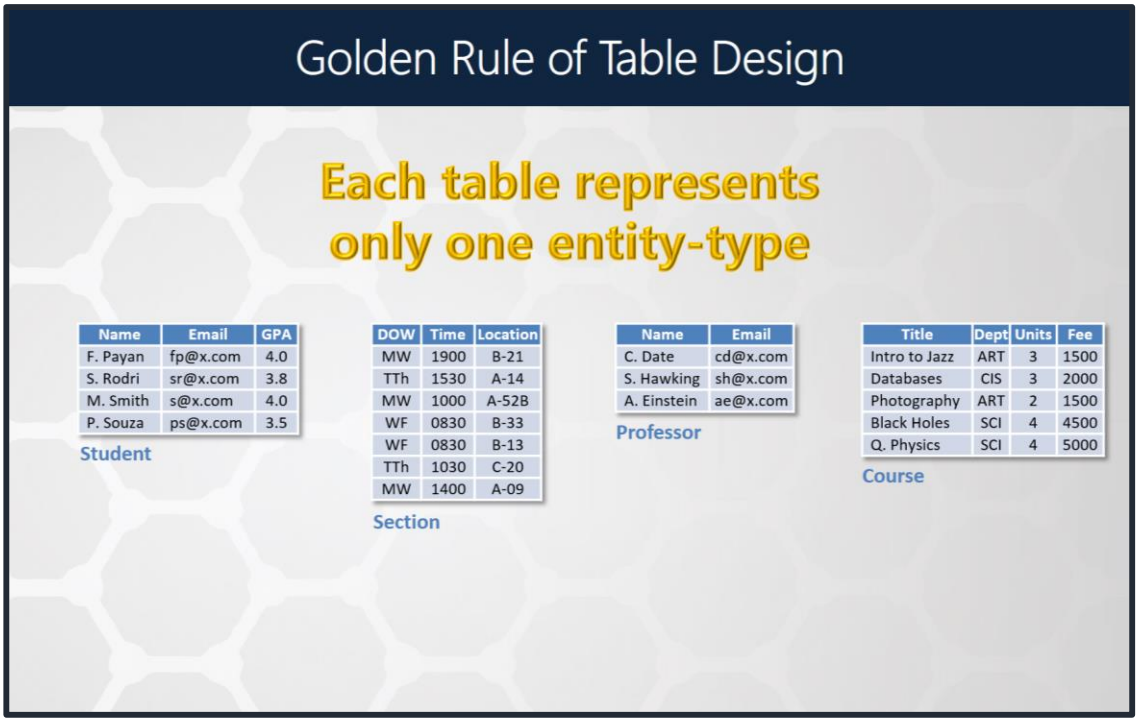
An insertion anomaly occurs when the database enforces an existence dependency that does not exist in the miniworld.

Deletion Anomaly

The deletion anomaly is the reverse of the insertion anomaly. Let's say that F. Payan and S. Rodri will not be able to attend their courses, after all. If we delete the first 2 rows in the table, we not only delete their enrollments, but we also delete the fact that we offer a course called Databases that belongs to the CIS department, is worth 3 units and costs $2,000. We also delete data about that one section and Professor Date.

A deletion anomaly occurs when deleting an entity in the database causes the unnecessary deletion of other entities along with it. Like it's half-brother the insertion anomaly, the deletion anomaly is also the result of a bogus existence dependency. Out goes an enrollment? OK, then out also goes the course, the professor, the section…

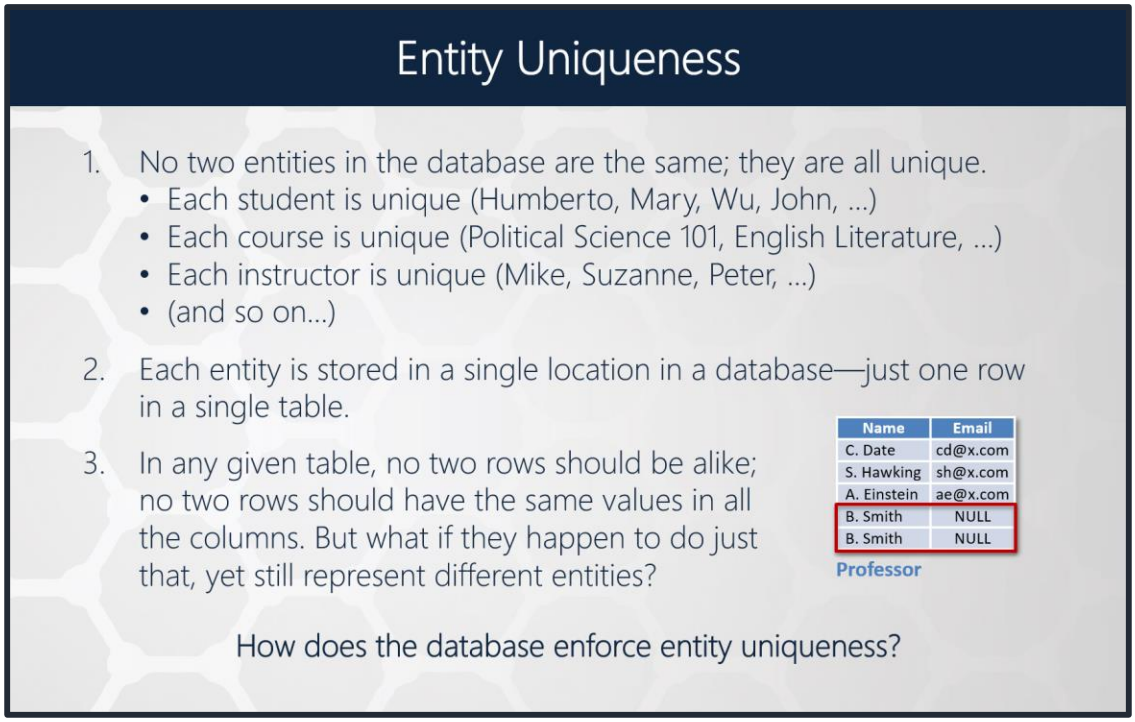
These three data anomalies are the direct result of including more than one entity-type in a single table.



This is the single, most important rule to follow when designing tables, because adherence to this rule virtually eliminates the possibility of anomalies. On the face of it, it might seem that this should be an easy rule to follow, but foreign entity-types have subtle ways of creeping into tables.

To ensure that the rule is being followed, database designers will often apply the *rules of normalization* to the tables in a database. Each rule is called a *normal form*. A database is normalized when it is in compliance with one or more normal forms. Tables that are not in compliance with a normal form can be put into compliance by decomposing the table in specific ways described by each normal form.

Normalization is beyond the scope of this course. For an authoritative discussion of normalization and the normal forms, please refer to Chris Date's book, "An Introduction to Database Systems."



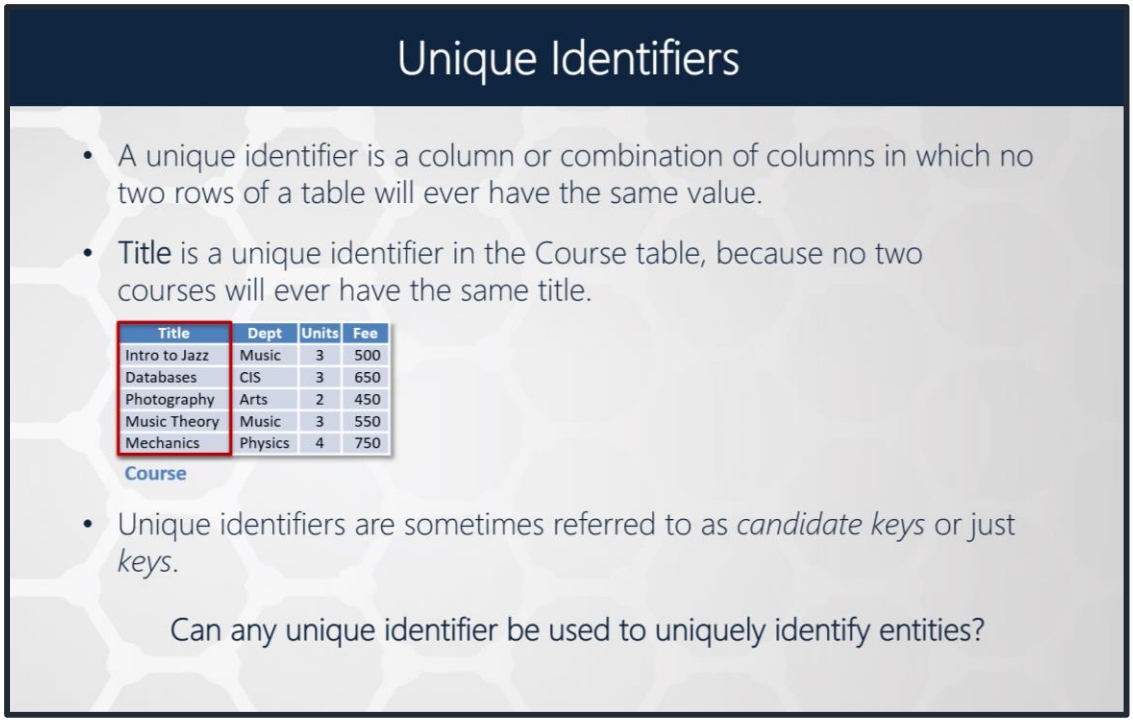
As you saw when you learned about the database building blocks, no two entities are alike; entities are unique, individual things. This is an important trait of entities and the database must be able to enforce it.

To begin with, each entity is stored in one, unique location in the database—a single row in just one table and nowhere else.

You also learned earlier that an entity is described by its attribute values. The Mountain-200 bike is different from the Race-150 bike. They are both bikes, so they have the same attributes; what is different about them—from the perspective of the database—is that the values in the attributes are different. The Name attribute of the first bike contains a different value (name) than does the same attribute for the other bike. The SafetyStockLevel attribute might have different values, and so on.

It's tempting to think at this point, that no two entities will have the same combination of values in their attributes, but as the slide above shows, it can happen! Apparently two professors have the same name (B. Smith) and, since we don't know their email addresses, we have two identical rows even though the entities they represent are distinct.

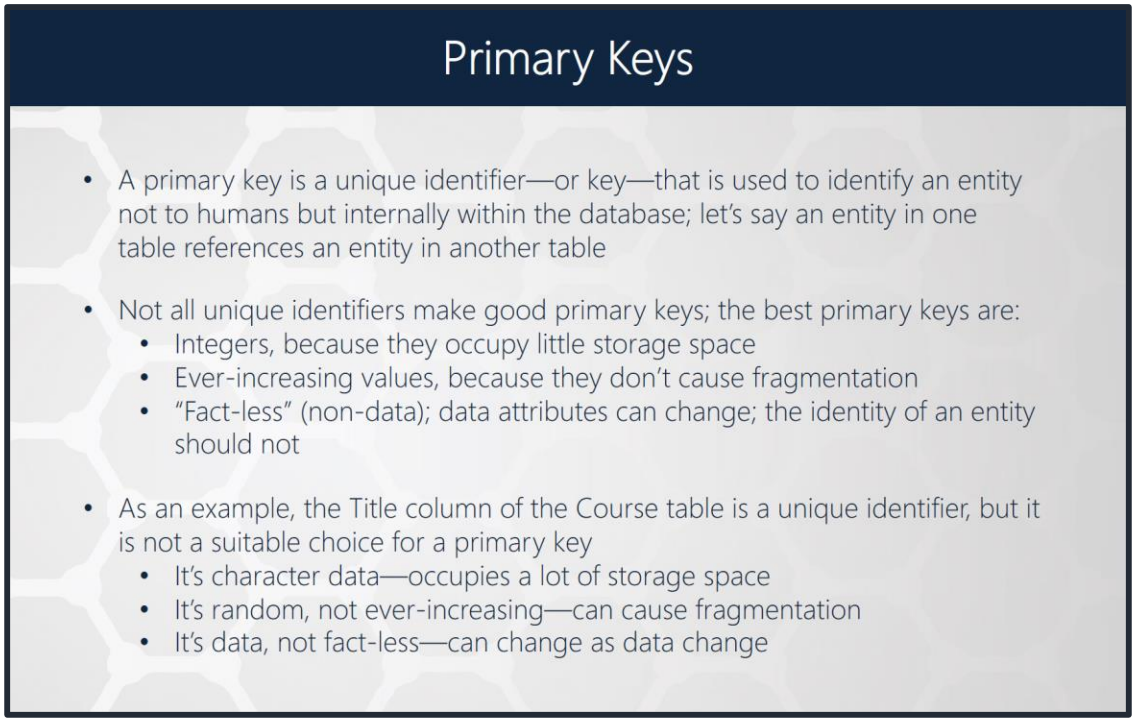
What is needed is an identifier of some kind—a value that is unique to each entity—very much like we each have a social security number.



Unique identifiers can often be found in the data, as the example of the Course table on the slide shows. In a Product table, a ProductName column is also a unique identifier. Names are not unique identifiers when it comes to people, but they are with most other entity-types. EmailAddress is another common unique identifier, and there are others.

Unique identifier is synonymous with *key* and *candidate key*.

If a table has a unique identifier, can we just use that to uniquely identify entities within the database?



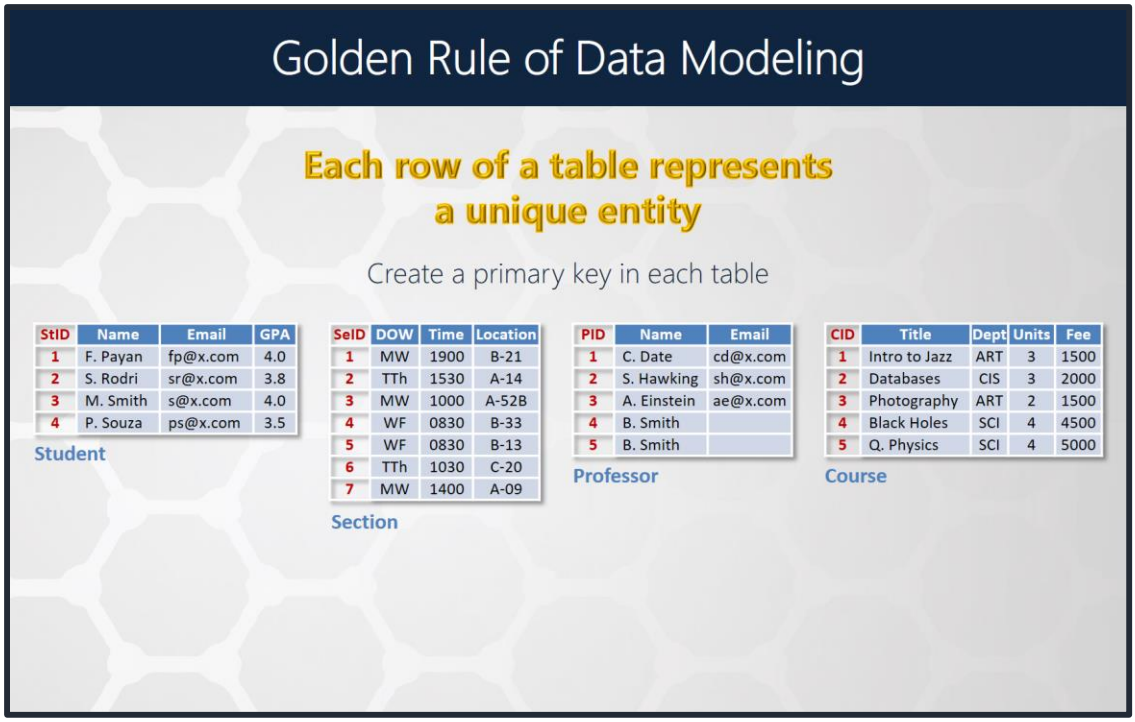
A primary key is a unique identifier that is used to identify entities within the processes of the database—it plays the role of the identity of an entity. As humans, we usually use a unique identifier that occurs naturally (is one or more attributes) in the data (*natural key*), like the course title. If you bump into someone in the hallway and they ask you what course you're taking, you're likely to answer with the course title. But, to the database, natural keys are poor choices for a few reasons.

First, natural keys may be of many data types. Some of those data types, like character, occupy a lot of storage space. We want to keep primary keys short because, as you will see, though we don't want to duplicate data, keys are another matter; keys must be duplicated in the database for the database to work. Integers are the best choice because they can represent large values and occupy relatively small amounts of storage space.

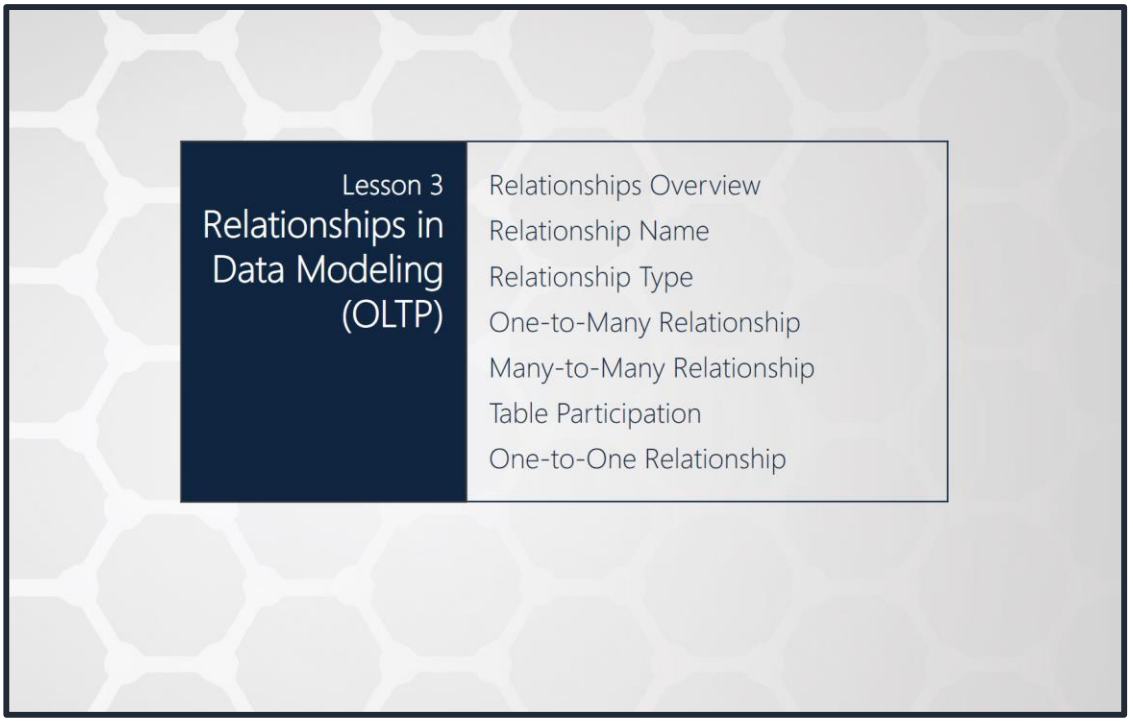
Second, primary keys should be ever-increasing values to prevent fragmentation. This means that every new entity inserted into a table must have a primary key value that is larger than the value of the primary key most recently inserted into the table. So, if the primary key value of the first entity inserted into the table is 1, the primary key of the second row must be greater than 1, the primary key of the third row must be greater than the primary key value of the second row, and so on. This is important because of the way clustered indexes work. Ever-increasing values prevent fragmentation of the clustered index. Fragmentation can slow down a database very noticeably.

Finally, data can change. We don't want the identity of an entity to change just because some of its data change. Let's say the title of a course is changed from "Introduction to Databases" to "Fundamental of Databases" and that nothing else changes about the course, including the outline and curriculum. It's the same course, just with a different name, now. If the title is used as a primary key, the database will respond as if it were a new course, disconnected from the records of what it now sees as the "old" course.

The best primary keys are artificial values, preferably integers, which have nothing to do with the data—just like your social security number.

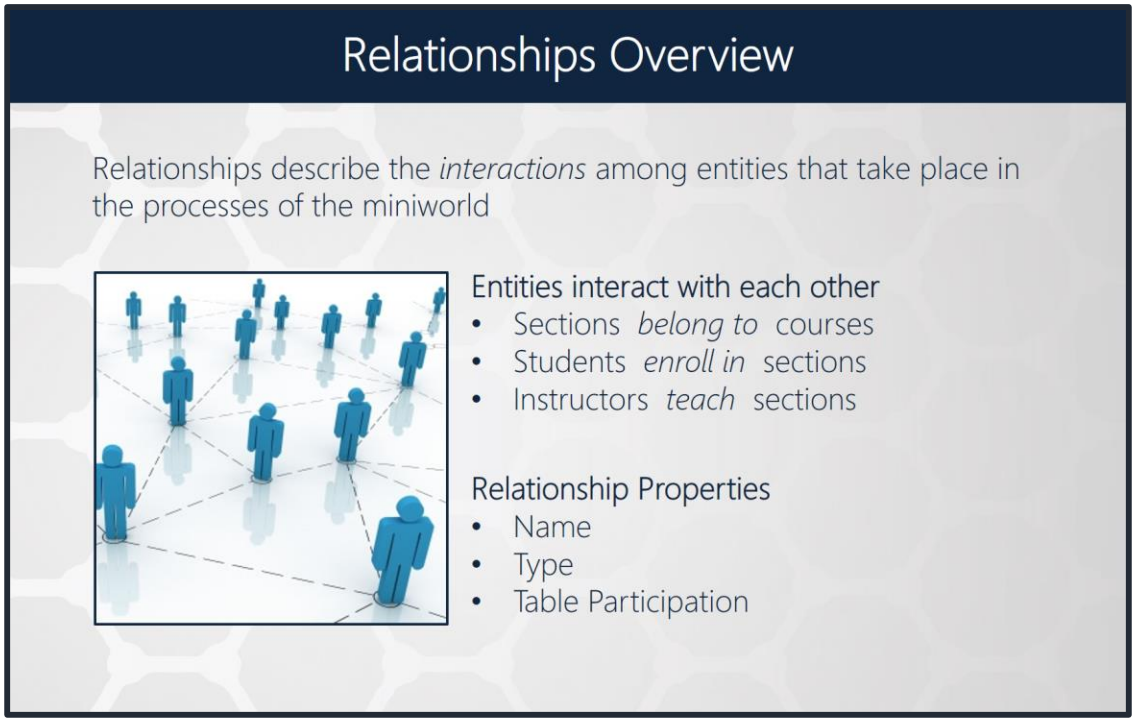


With the addition of primary keys to the tables, the two professors with the same names (B. Smith) and for whom we don't have email addresses, no longer have the same values in all the columns and are now recognized as distinct entities in the database.



This lesson is the core of this module. It is here in lesson 3 that you will understand how a database is organized to represent a model of a miniworld in a way that allows SQL to mainpulate it, both for maintenance and analysis.

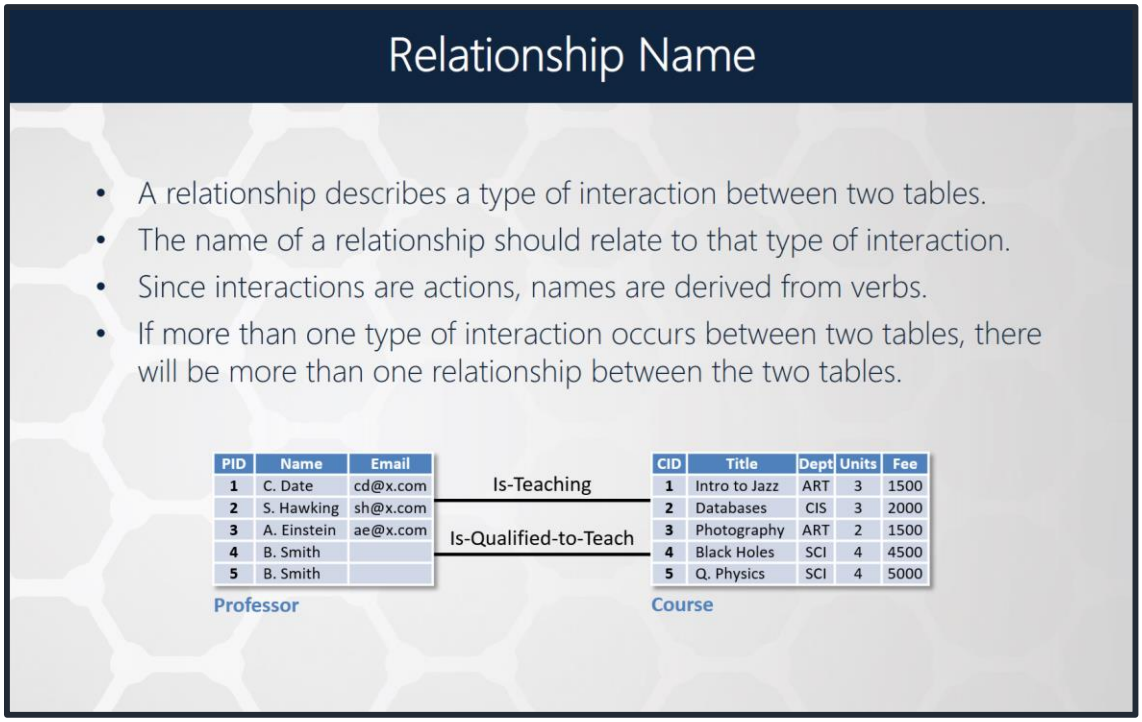
The foundation you will build in this lesson will form the basis for a lot of things you'll be doing in this course.



The entities in the mini-world don't just sit quietly in tables; they interact with each other in the processes of the mini-world. Products are sold, job candidates submit resumes, students enroll in courses, and so on.

Interactions among entities are represented in the database as *relationships*. To show that a product is ordered, a relationship between that product and an order is established in the database. To show that a course is being taught in a particular section, a relationship is established between that course and its section.

Relationships between members of one entity-type and another have two other properties that will be useful in this course: Table cardinality and participation will help write queries that join tables.



Fundamentally, a relationship between tables describes a way in which those two tables interact with each other. If they interact in more than one way, then there can be more than one relationship between the two tables!

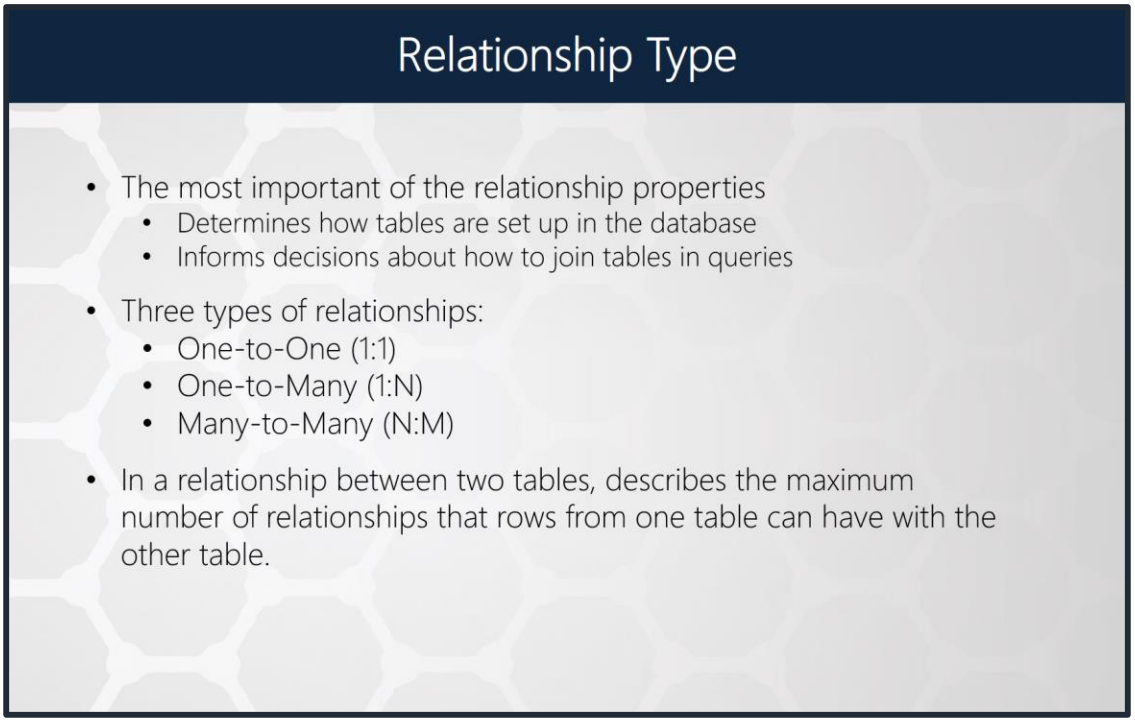
While the names of relationships are often overlooked, they are important because they tell us something about the nature of the relationship. The name of a relationship should describe the interaction that takes place between the participating entities. Because relationships represent interactions, they are commonly named using verbs. The name of the relationship describes the interactions of one of the tables with respect to the other. Here are some examples of relationship names:

Courses ARE OFFERED IN Sections
Sections BELONG TO Courses
Relationship Name: AreOfferedIn or BelongsTo

Professors TEACH Sections
Sections ARE TAUGHT BY Professors
Relationship Name: Teaches or IsTaughtBy

Customers PLACE Orders
Orders BELONG TO Customers

Relationship Name: Places or BelongsTo

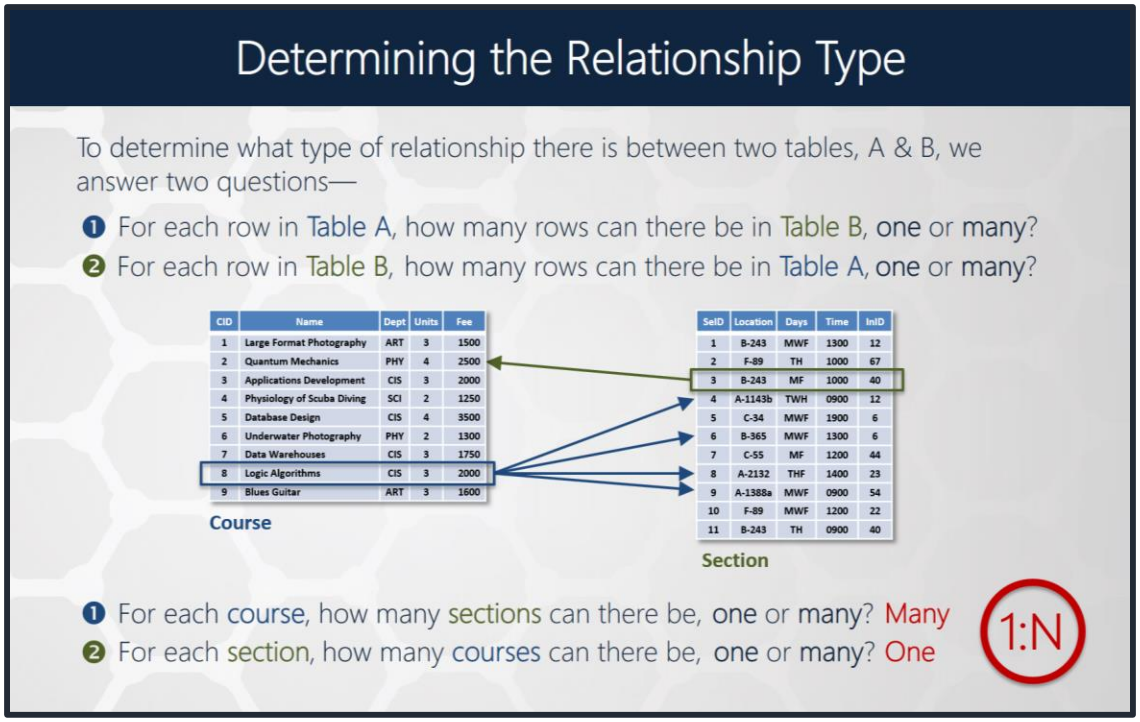


The relationship type is the most important property of a relationship because it determines how the relationship will be represented in the actual database—which table has the foreign key, which table is referenced by the foreign key, and so on, as you are about to see. You will also use the relationship type whenever you join tables in a query.

There are three different types of relationships that can exist between two tables. In the underlying mathematics, they are cardinality ratios, which explains their names as shown on the slide.

So, what is the relationship type, exactly? It is the ratio between the maximum number of rows to which a row in one table can be related in the other table and the same number from the perspective of the other table.

Here, let's look at how to determine the type of relationship between two tables so you can see what that gobbledygook means. A picture is worth a thousand words…



Let's say you've determined that two tables, say Table A and Table B, are related (the entities in Table A interact with the entities in table B). In order to properly represent the relationship between the two tables in the database, you will need to know the relationship type—whether it is 1:1, 1:N, or N:M. How do you find out?

You answer two questions. Here's the first one:

1. For each row in Table A, how many rows can there be in Table B?

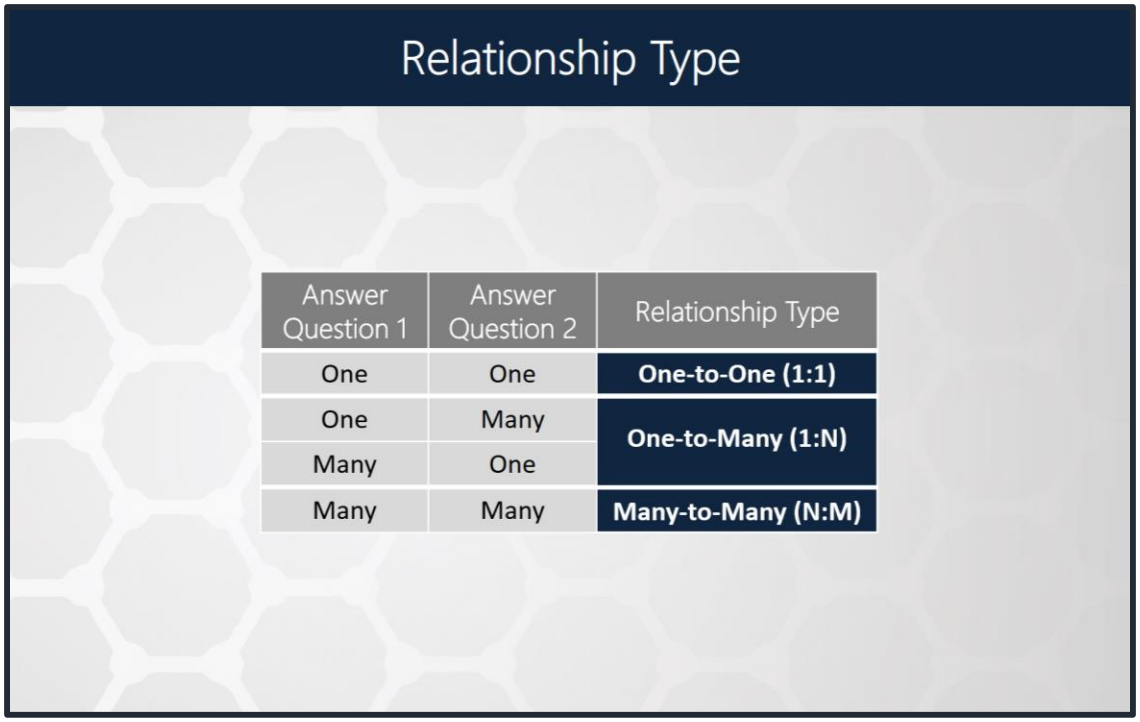
The answer to this question must be either one or many, in which "many" means "more than one." That is the difference that determines how tables are set up in the database. Notice also the use of the word "can." That implies how many there can be *at most*. Is it one or more than one? It has to be at least one; otherwise there is no relationship? If the maximum number of relationships that a row can have with rows in the other table is none, then there is no relationship.

The second question is almost identical to the first one but from the perspective of the other table, so you'll see the tables swapped:

2. For each row in Table B, how many rows can there be in Table A? Again, one or many.

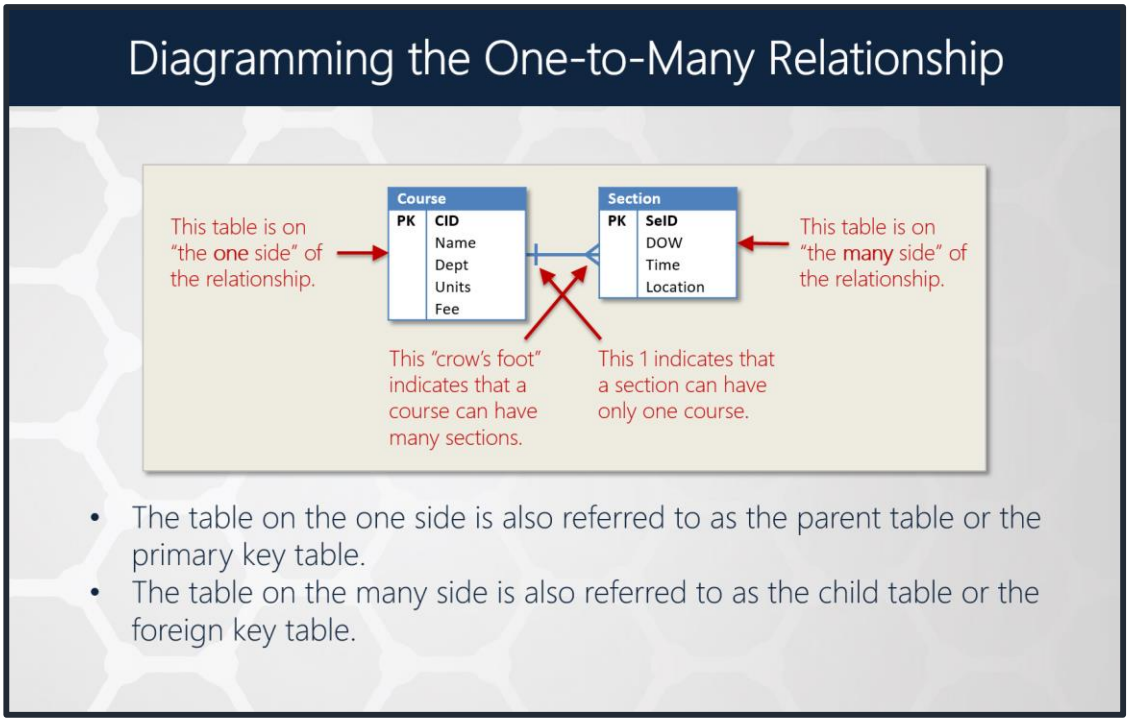
If the answer to both questions is one, then the relationship is 1:1. If the answer to one question is one and to the other it is many, then the relationship is 1:N. And if it is many for both questions, then the relationship is N:M.

In the example on the slide, the relationship is determined to be 1:N. As you'll see, this is the most important type of relationship; it wouldn't be inaccurate to regard the other two as special cases of the one-to-many.



This slide summarizes the discussion of the previous slide.

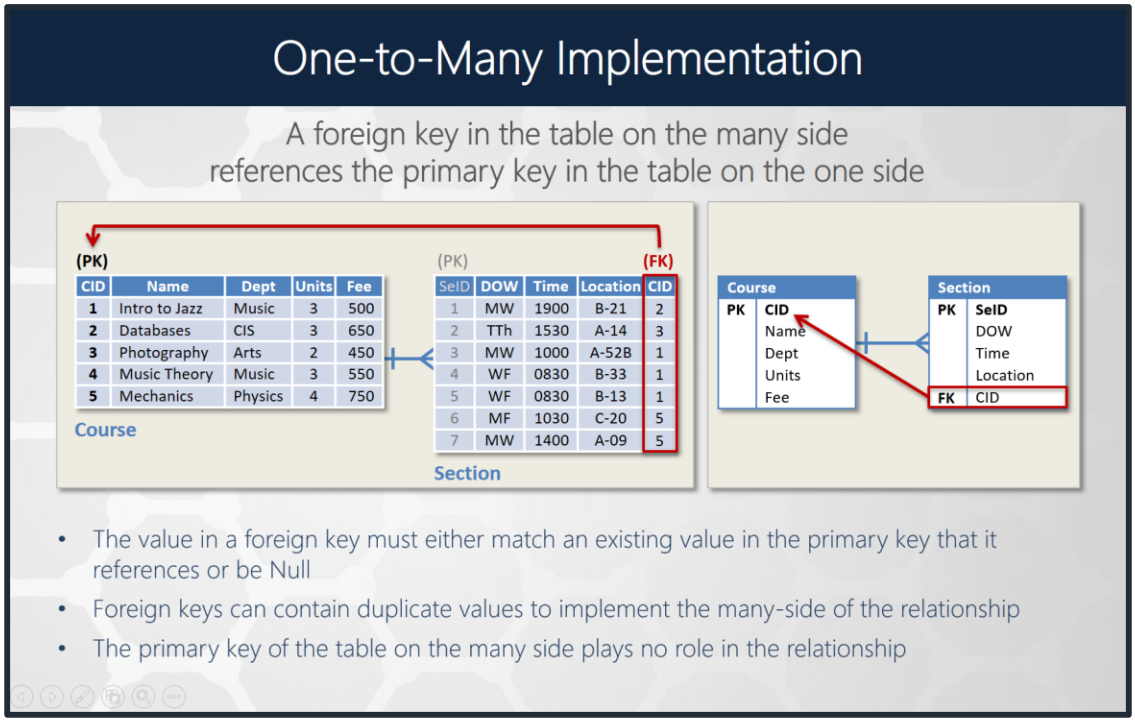
The relationship type is derived from the answers to the two questions. If the answer to both questions is one, then the relationship is one-to-one. If the answer to one of the questions is many and to the other it is one, then the relationship is one-to-many. And if the answer to both questions is many, then the relationship is many-to-many.



While there are many notations used to diagram databases, this course emphasizes the crow's foot notation because it is the richest in information.

To diagram a relationship between two tables, we draw a line between them. The symbols next to each table describe the cardinality of the other table. In the diagram you see on the slide, the cardinality of the Course table is many, which is represented by a crow's foot next to the Section table. The cardinality of the Section table is one, which is represented by a symbol for one next to the Course table. We read the diagram by saying that a course can "point to" many sections whereas a section can point to no more than one course.

We say that the Course table is *on the one side* of the relationship and the Section table is on the many side. The table on the one side is also referred to as the *parent* table and the table on the many side is also referred to as the *child* table. And for reasons that will soon become clear, the table on the one side is also referred to as the *primary key* table, while the one on the many side is also referred to as the *foreign key* table.



It is one thing to draw lines between boxes in a diagram, but how are relationships implemented in the database?

In this example, a course can have many sections. You might imagine a column in the Course table called Sections in which all the sections for a course would be listed, but that would be a multivalued attribute. Multivalued attributes are not allowed for data integrity reasons.

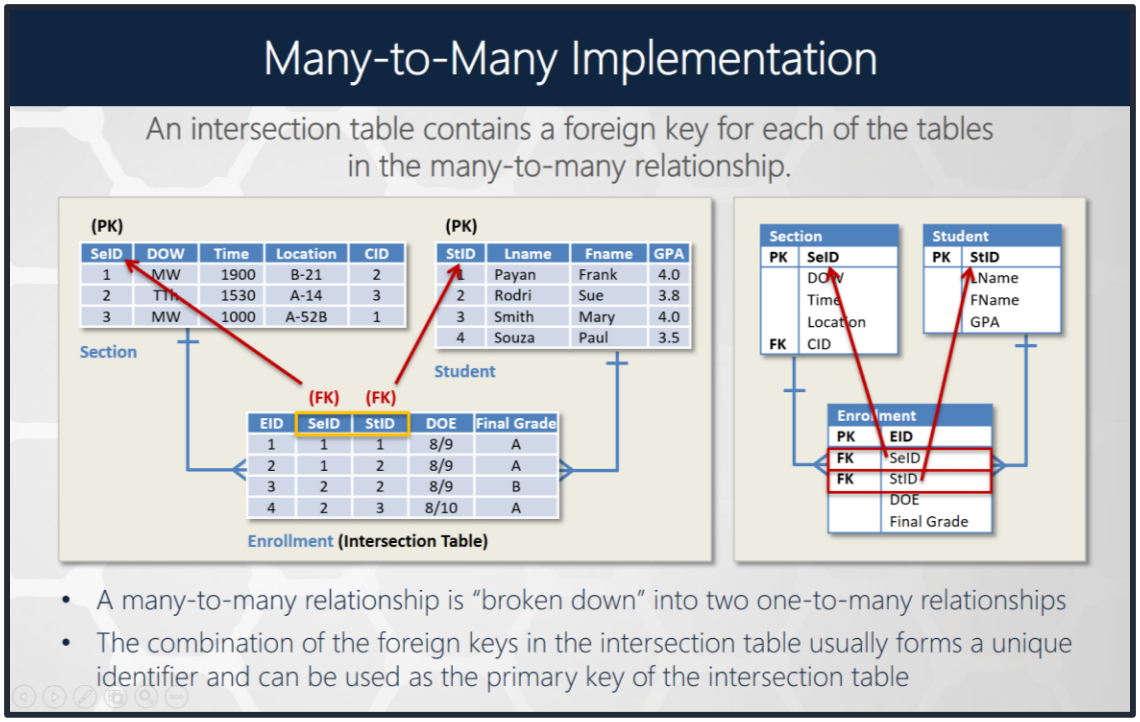
A better place to represent the relationship between courses and sections is in the Section table. Since a section can belong to only one course, all that is needed is an extra column in the Section table that holds the CID of that course. Each row, or section, would be effectively "stamped" with the ID of its owner. That "stamp" column is called a *foreign key*.

In general, a one-to-many relationship is represented in the database with a foreign key in the table on the many side that references the primary key of the table on the one side.

Here are a few things to note about foreign keys:

- Foreign keys are not unique; they can contain duplicate values. This is necessary since they are on the many side of their relationships. In the example on this slide, sections that belong to the same course will all have the same CID value in the CID foreign key column.

- Nulls are allowed in foreign keys.



The many-to-many relationship is not directly represented in the database; it is broken down into two one-to-many relationships in which the intersection table is on the many side of both. In the diagram on the slide, the Enrollment table is the intersection table.

Notice the two foreign keys in the intersection table. SectionID references the primary key of the Section table and StudentID references the primary key of the Student table. Each time a student enrolls in a section, a row is created in the intersection table that includes both a StID identifying the student and a SeID identifying the section.

In the first row of the intersection table, student 1 enrolled in section 1. In the next row, another student, student 2, enrolled in the same section; that's one section, many students.. In the third row, student 2 also enrolls in section 2. The same student now has many sections.

The intersection table shows that a section can have many students and that a student can be enrolled in many sections. It implements the many-to-many relationship between the Section and Student tables.

Naming Intersection Tables

Understanding how intersection tables are named can help us find them and sheds some light on how tables are used.

The intersection table between Section and Student is also an entity-type in the mini-world. Each row in the intersection table represents an enrollment entity with its own

attributes (DOE, Final Grade, and possibly others) and so the intersection table is named after that entity-type—in this case, Enrollment.

This isn't always the case. There are times when an intersection table exists solely for the purposes of intersecting a many-to-many relationship and does not represent an entity-type. Since an intersection table implements a relationship, the name of the relationship can be used. As we saw earlier in this module, relationship names are verbs. It is good form to use nouns for table names because, even though the table might not represent an entity-type in the mini-world, it represents an entity-type in the database—in this case a relationship entity-type.

Here's an example: Between the Course and Professor entity-types, there is a many-to-many relationship called Is Qualified to Teach. A professor can be qualified to teach many courses and each course can be taught by many qualified professors. In this case, the name of the intersection table might be Qualification—each row is a qualification entity that relates a professor to a course.

If all else fails and a descriptive name cannot be found for an intersection table, then the names of the original tables can always be combined to form one name for the intersection table. An intersection table between the Section and Student tables could be called SectionStudent.

Intersection Table Key

Usually but not always, the combination of the foreign keys forms a unique identifier, or key, in the intersection table. This is the case in the example shown on the slide as long as each school term gets its own enrollment table. In that case, a student enrolls in a section only once (in a term) and the combination of SectionID + StudentID is unique.

But if the same enrollment table is used from term to term, and if sections and their ids are usually the same from term to term, a student might enroll in sections by the same id in different terms. In that case, the combination of the two foreign keys would not be unique, though the combination of both foreign keys + DOE (Date of Enrollment) would be unique.

We can extend that and include a Term table in the database with certain attributes of its own and then include a third foreign key in the intersection table that would reference the Term table, effectively creating a ternary relationship (as opposed to the binary ones we've been working with). Now we're back to being able to say that the combination of all the foreign keys forms a unique identifier.

Here's another example: We start with two tables—an Employee table and an Equipment table. An employee can use many equipment entities and each equipment entity can be used by many employees, so the relationship between the two is many-to-many. Let's call the intersection table, EquipmentCheckout. Here are its attributes (the primary key is underlined):

EquipmentCheckout (ChkID, EmployeeID, EquipmentID, Date Assigned, TimeOut, TimeIn).

Because an employee can check out the same equipment multiple times, the combination of the two foreign keys is not unique. But here again there is an implied third entity-type, Time. We could conceivably create a Time table in which each row represented a date and time. A third foreign key referencing the Time table could then be added to the EquipmentCheckout table (OLAP databases commonly implement such time tables).

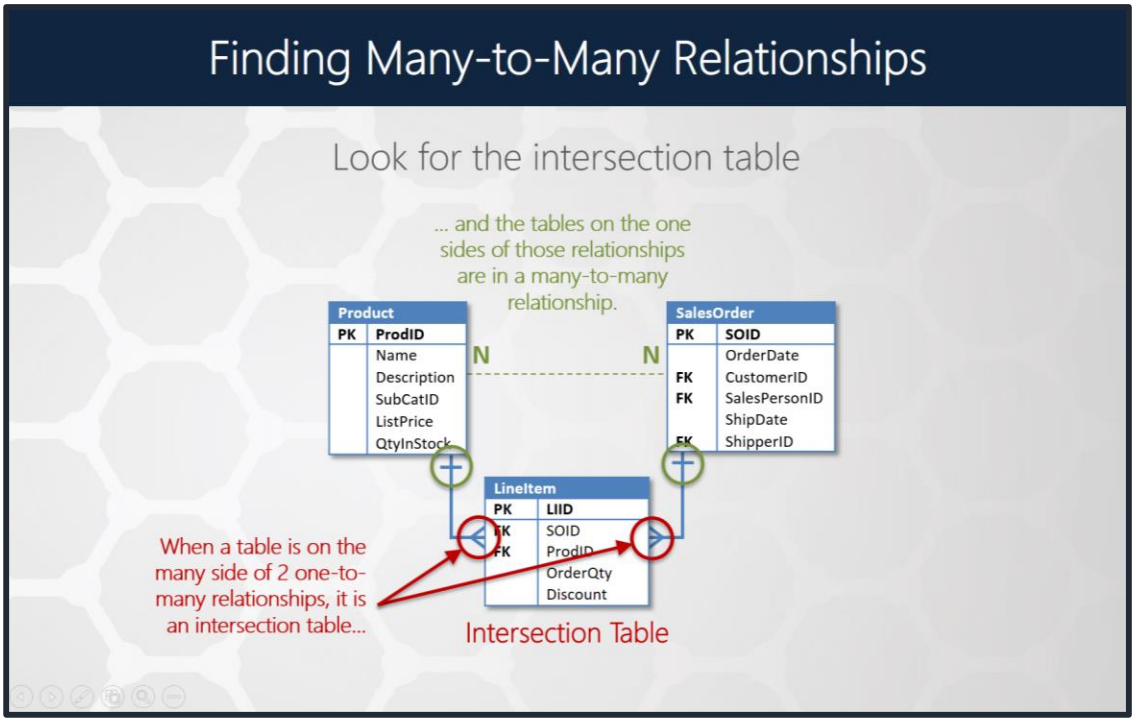
When a combination of the foreign keys is not unique, it is usually because there is an implied entity-type that is not being referenced by a foreign key in the intersection table, usually because other attributes of that entity-type are not of interest.

Intersection Table Primary Key

Some database designers willingly implement the combined foreign keys as a composite primary key of the intersection table. There is only one primary key (there can never be more than one primary key in any table) but it is made up of more than one column.

Other designers argue that this adds an unnecessary layer of complexity. If a relationship is established at a future date between the intersection table and some other child table, that child table will need a foreign key that references the primary key of the intersection table. It works if the primary key is a composite, but it requires jumping through a few more hoops, and queries that join the two tables will have the same hoops to jump through.

The alternative is to create an artificial primary key, say an EnrollmentID and also a unique identifier to enforce the uniqueness of the composite foreign keys.



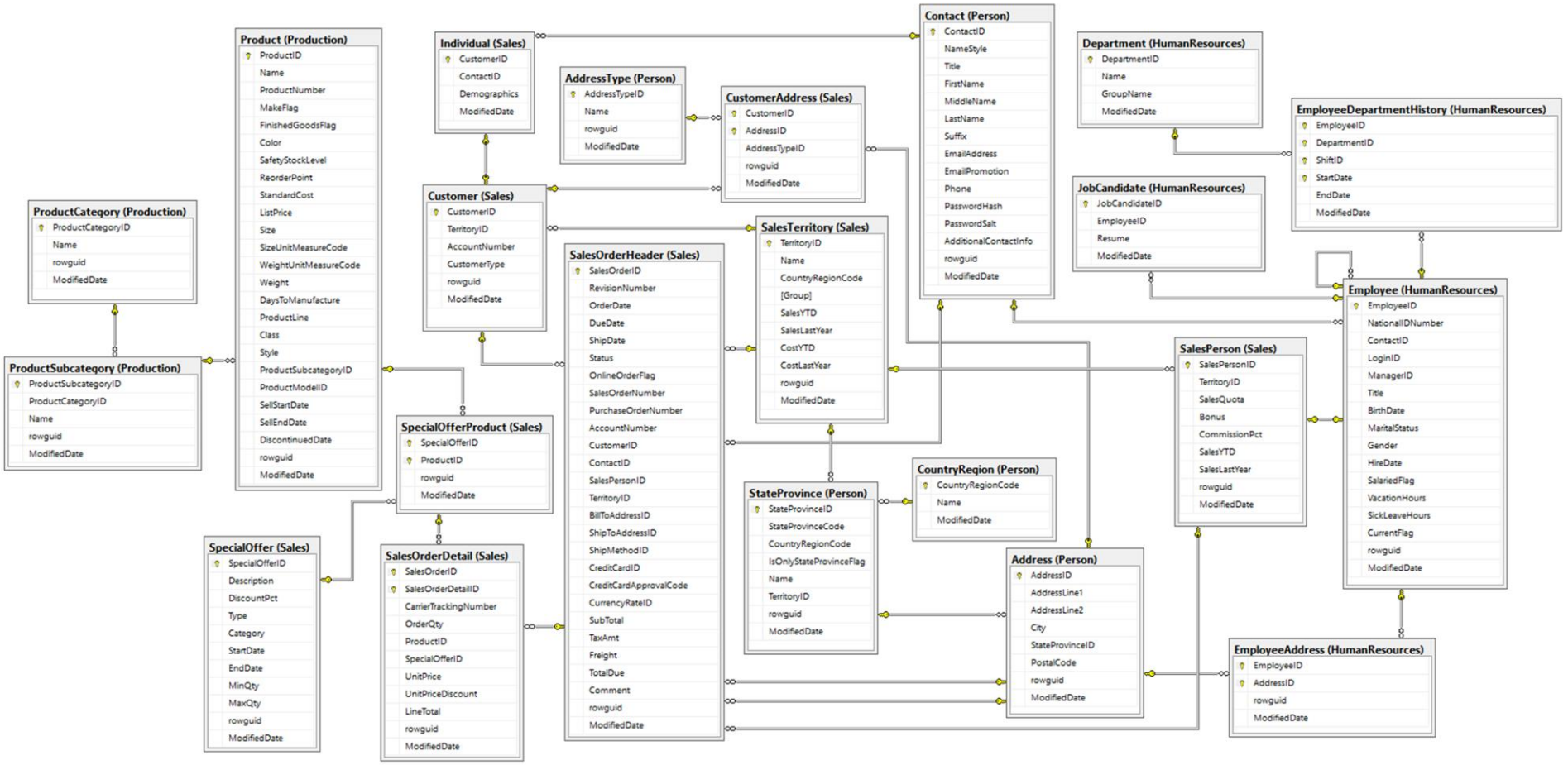
Because many-to-many relationships are not directly represented in a database and are instead broken down into one-to-many relationships, they are sometimes hard to find as they are "camouflaged" among other one-to-many relationships. Have a look at the partial diagram of the AdventureWorks database below. Notice that almost all relationships are one-to-many! (This diagram was generated in SQL Server; SQL Server uses a key symbol for the table on the one side and an infinity symbol for the table on the many side).

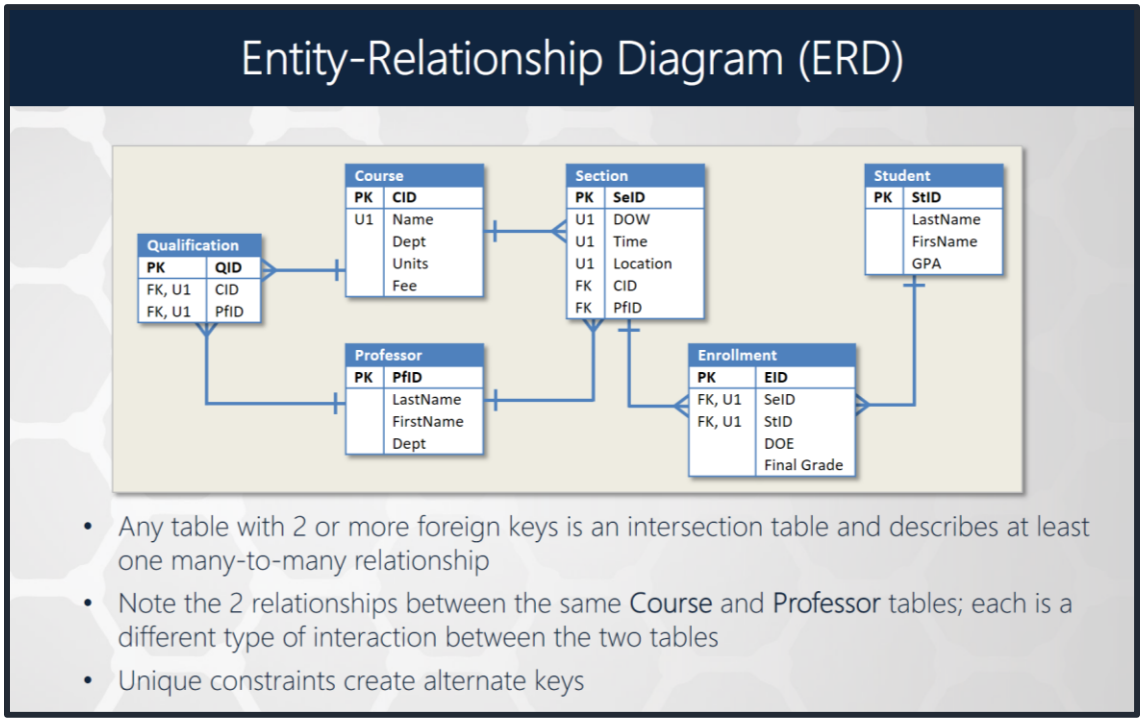
To find many-to-many relationships in the diagram, look for the intersection tables. One way to find them, though not a very reliable one, is to look for composite names. In the diagram on the next page of the AdventureWorks database, the CustomerAddress and EmployeeAddress tables are dead give-aways; they're named using the classic "combined name" for intersection tables.

A more reliable way to find intersection tables is to look for a table that's on the many side of two one-to-many relationships. The tables that are on the one side are in a many-to-many relationship. Can you find more in the diagram on the next page?

Here's one to get you started: The SalesOrderDetail table is an intersection table; it's on the many side of 2 one-to-many relationships. The tables on the one side of those relationships are the SalesOrderHeader and SpecialOfferProduct tables. So, those two tables are in a many-to-many relationship; a sales order can contain many products on special offer, and each product on special offer can be sold in many sales orders. How many more can you find?

Partial Schema of the AdventureWorks Database





Here is the diagram of our database so far. Notice that we have also added the Professor table in a one-to-many relationship with the Section table, since a professor can teach many sections, and every section is taught by only one professor.

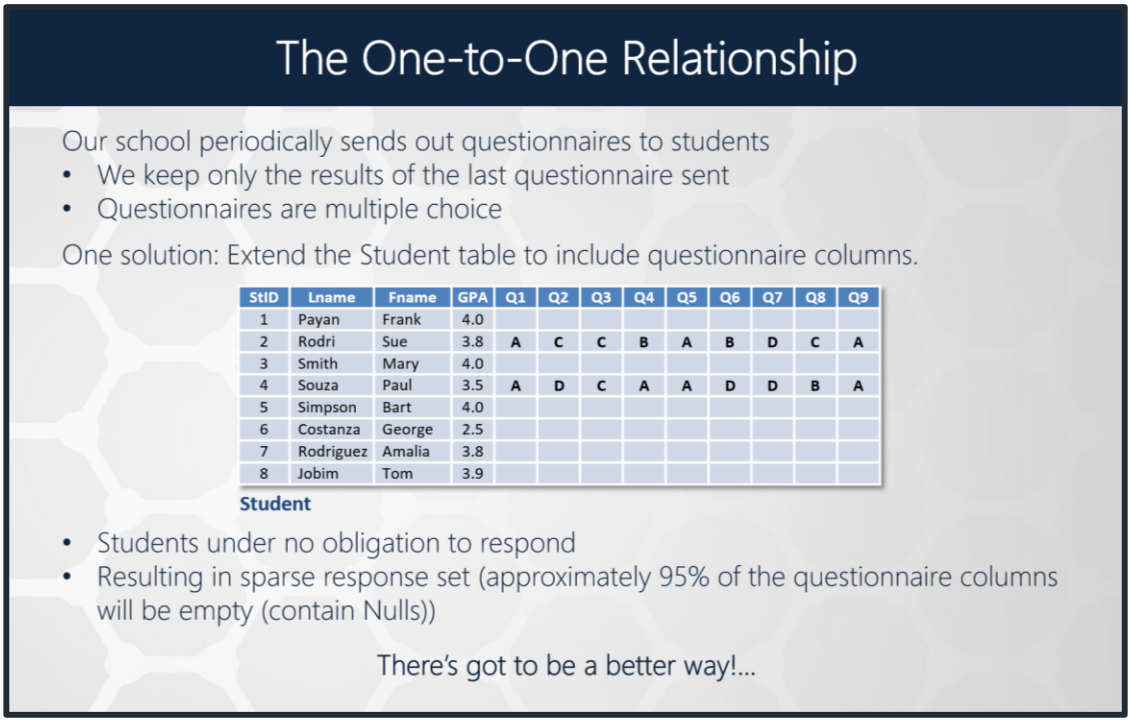
The new one-to-many relationship between Professor and Section reveals that Section is an intersection table and that there is a many-to-many relationship between Course and Professor. That many-to-many relationship shows which professors are teaching which courses and which courses are being taught by which professors. We can call that relationship the IsTeaching relationship.

Notice that the IsTeaching relationship does not tell us which professors are *qualified* to teach which courses; just because professor A is not teaching course X (in any sections) doesn't mean that professor A is not qualified to teach course X.

This means that there is an additional many-to-many relationship between the Course and Professor tables! We can call it the IsQualifiedtoTeach relationship. In the diagram shown on the slide, the Qualification table is the intersection table in that relationship. Each row in the Qualification table is a qualification entity.

It is perfectly normal for two tables to have multiple relationships between them; each relationship represents a different type of interaction between the two tables.

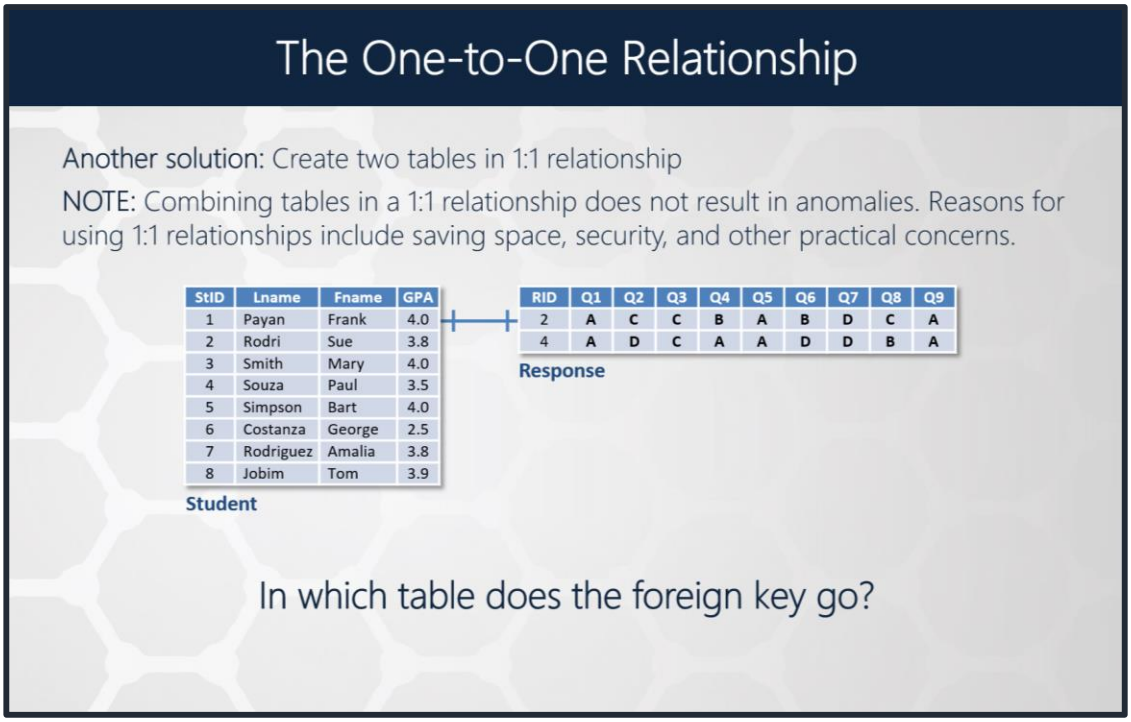
The only other relationship type we have not yet discussed is the one-to-one. Let's have a look at that now.



Every once in a while, the school sends a questionnaire to students. Since all questions are multiple choice and we keep only the results of the last questionnaire sent, one way to represent questionnaire responses would be to add a column for each question to the Student table.

The problem with this approach is that, because students are not required to respond to the questionnaires the school sends them, we can expect a very low response rate of somewhere between 3% and 5%. This means that the questionnaire columns for 95% of the rows will be empty.

There's a better way to do this…



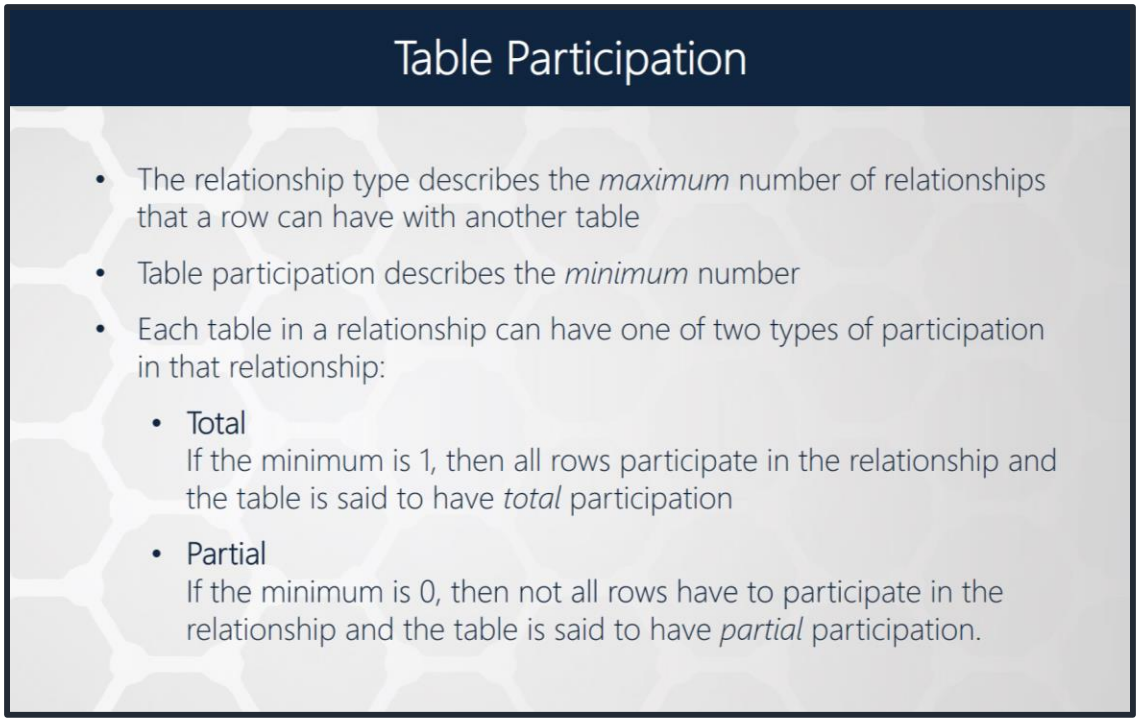
By using a separate response table, we eliminate the need to store so much empty space.

Notice that, unlike 1:N and N:M relationships, the 1:1 relationship is not used to prevent anomalies; combining two tables that are in a 1:1 relationship into one table does not result in data duplication since each row in one table can be associated with only one row in the other table!

The reasons we use 1:1 relationships are more practical in nature and include things like conservation of storage space, securing an entire table differently from the other rather than having different security structures on different columns, and so on.

But we're not out of the woods just yet! How do we implement the 1:1 relationship? Both the 1:N and N:M relationships were implemented using a PK/FK pair. The same applies in the case of the 1:1. So, where do we put the foreign key? In the 1:N relationship it goes in the table on the many side. In the N:M relationship the foreign keys go in the intersection table—again, the table on the many side.

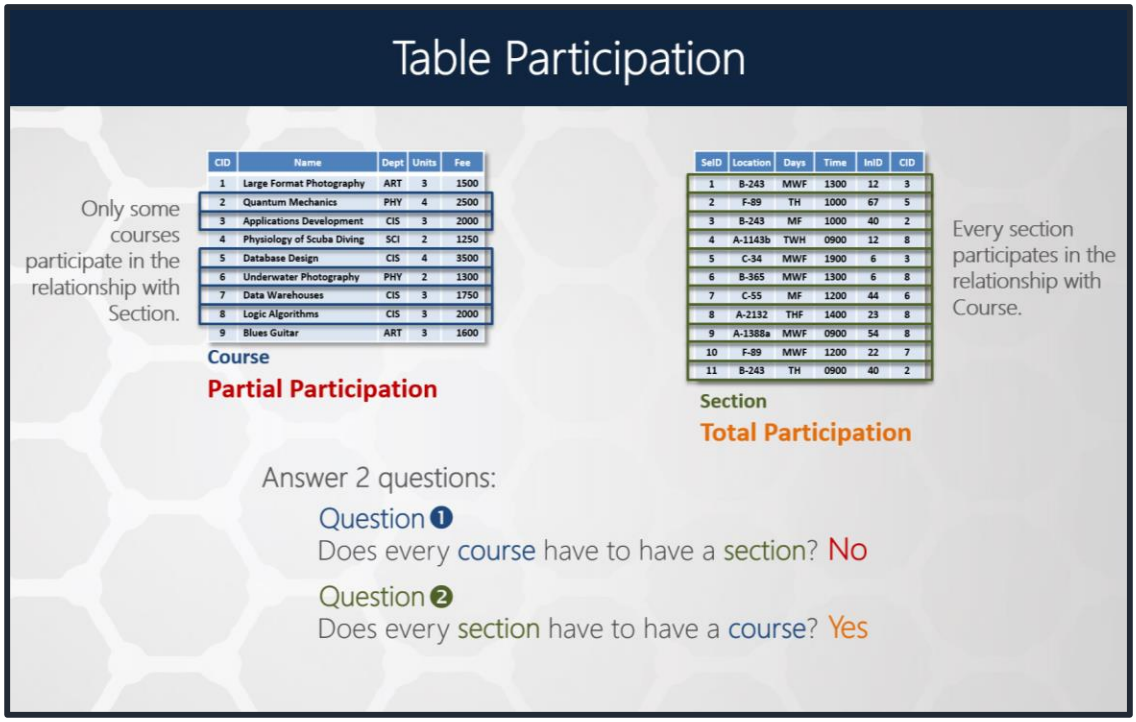
The 1:1 relationship does not have a table on the many side. We need another criterion.



While the relationship type describes the maximum number of relationships that a row in a table can have with another table, table participation describes the minimum. The maximum was expressed as either one or many. The minimum is expressed as either zero or one.

If the minimum number of relationships that a row in one of the tables can have is zero, then not all rows in that table participate in the relationship and so that table is said to have partial participation in the relationship. On the other hand, if the minimum is one, then there has to be at least one relationship for every row; that means that the entire table participates in the relationship and so that table is said to have total participation in the relationship.

Let's see an example of this.

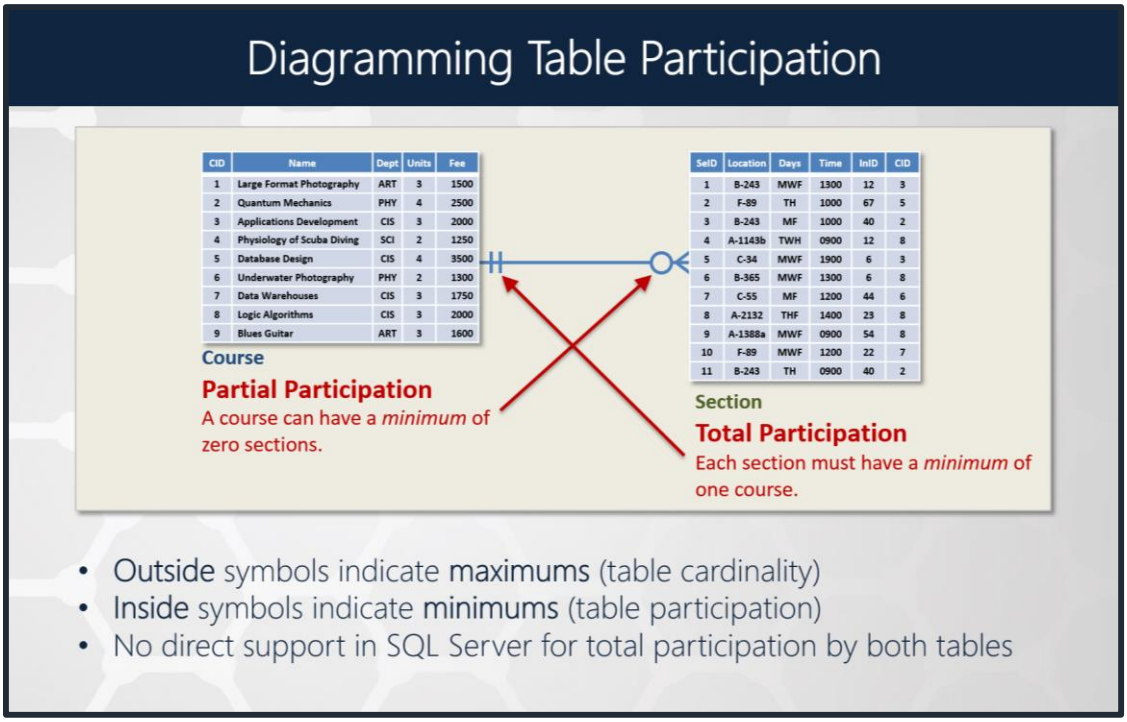


In our example of the relationship between the Course and Section tables, not every course has to have a section. The courses that do are outlined in blue. You can see that courses 1, 4, and 9 do not have sections. The participation of the Course table is partial, because only a part of the table (only some of its rows) participate in the relationship.

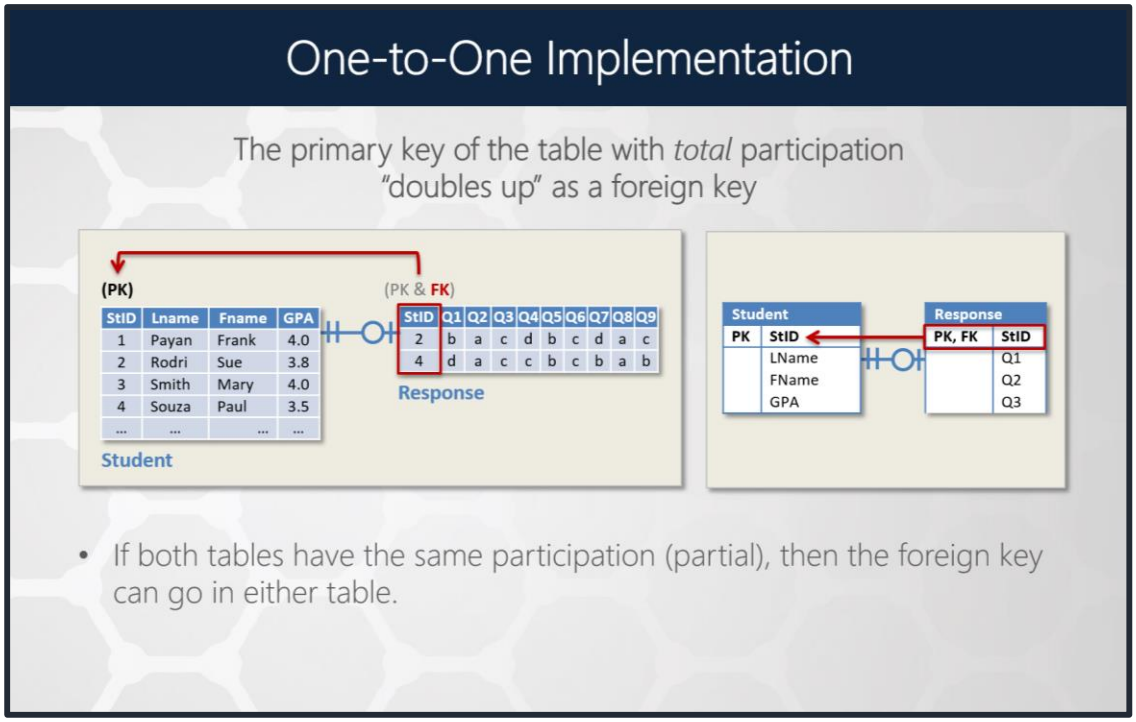
On the other hand, the participation of the Section table is total, because every section must have a course (a section is an offering of a course, so the existence of a section without a course would be an absurdity). The entire Section table (all of its rows) participates in the relationship.

As with table cardinality, table participation can also be described in terms of the cardinalities of the ρ sets of the table. Whereas table cardinality is the *maximum* cardinality of the ρ sets (1 or many), table participation is the *minimum* cardinality (0 or 1). The minimum number of rows in the Section table to which a row in the Course table must be related is zero. That's partial participation. The minimum number of rows in the Course table to which a row in the Section table must be related is 1. That's total participation.

As with table cardinality, table participation is not determined based on any particular state of the data but by considering all possible states. We can imagine a term in which all courses are offered, but that would not make the participation of the Course table total, because there can be courses that are not offered in another term (another state of the data).



To diagram table cardinality, we used symbols on the relationship line between the two tables. Table participation is also diagramed by using symbols on the relationship line; the difference is that the symbols for cardinality are on the "outside" of the line (closest to the tables), whereas the symbols for participation are on the "inside" of the line (farthest from the tables). It the participation of a table is total, we use a 1 symbol, and if it's partial, we use a zero symbol as you see in the diagram on the slide.

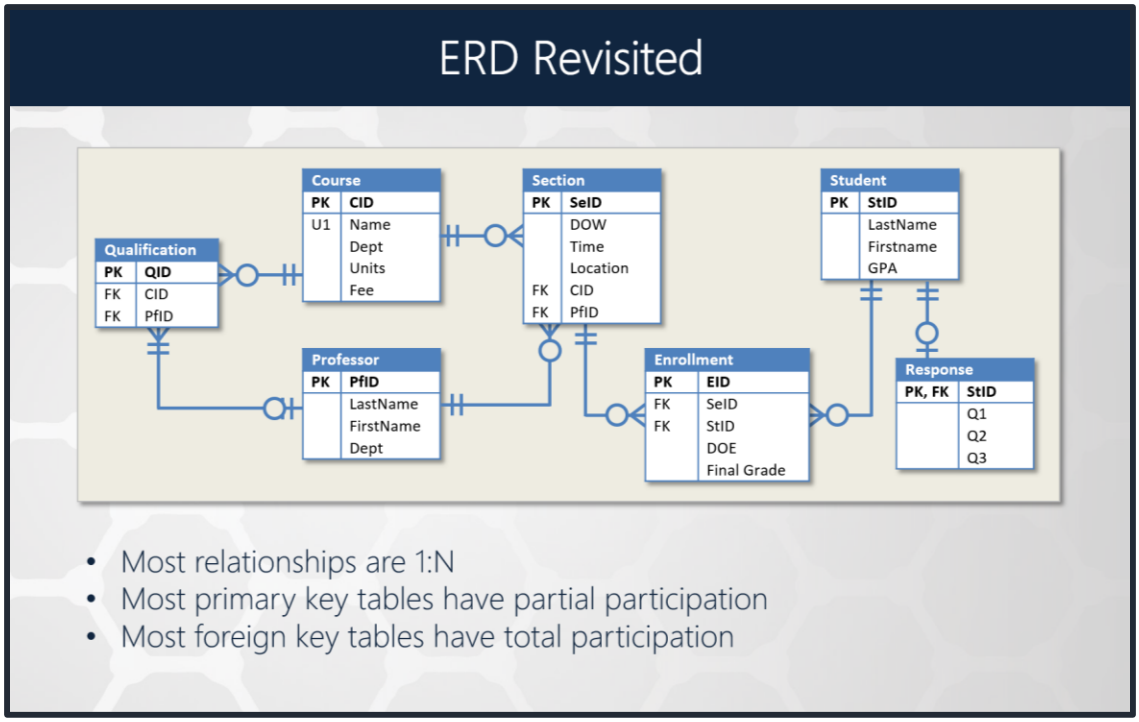


We now have a new criterion that can help us decide in which table to put the foreign key.

The Student table has partial participation in the relationship with Response, so if we put it in there, the rows that don't participate in the relationship will have Nulls in the foreign key. If one of the tables has total participation in the relationship, it's best to put it there.

The Response table has total participation in the relationship, so if we put the foreign key in the Response table, there will be no nulls in it since every response "points to" a student.

If both tables have the same participation, it doesn't really matter where we put it. Choose the one you expect to have the least amount of nulls; otherwise, choose one arbitrarily.



We can now add the 1:1 relationship and table participations to our ER diagram.

Notice that most relationships are 1:N and that 1:1 relationships are rare by comparison.

Notice also that in most 1:N relationships, the table on the 1 side has partial participation and the table on the many side has total participation. This is usually the case in most databases, but it's not always the case.

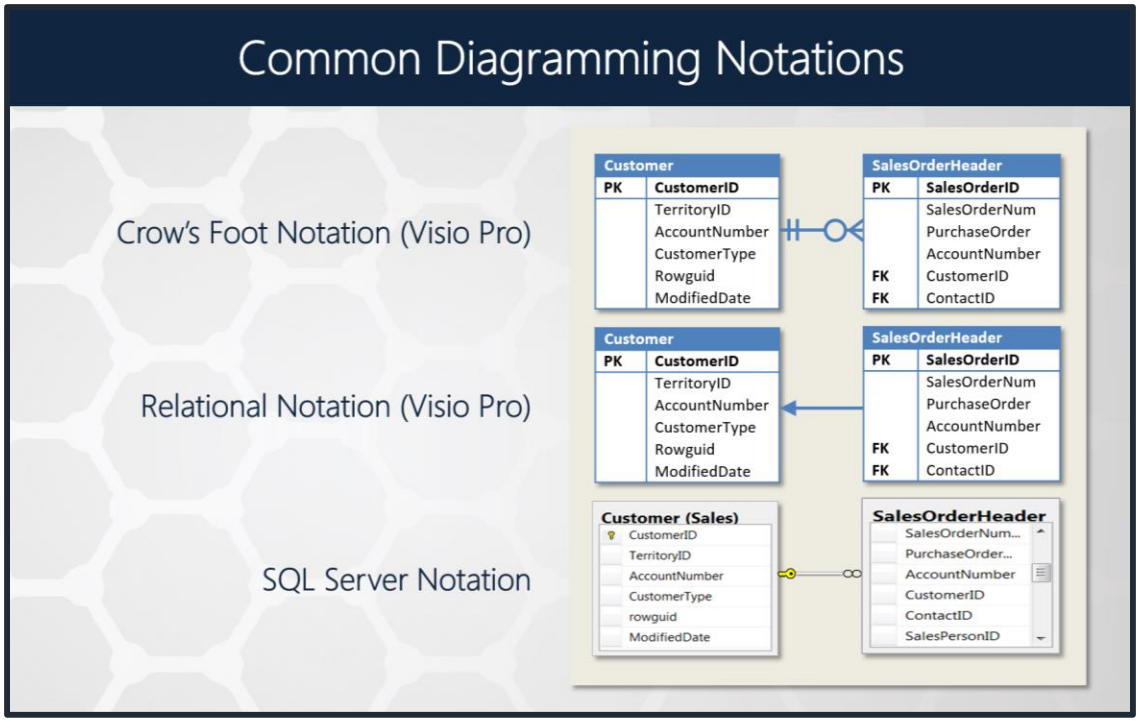
Consider a table called Equipment that is related to the Section table. Each equipment entity can be assigned to a section. In this example, the participation of both tables is partial; not all equipment is assigned and not all sections have equipment assigned to them. You can see all of this in the SeID foreign key in the Equipment table.

SeID	DOW	TIME	LOCTN	CID
1	MW	1900	B-21	2
2	TTh	1530	A-14	3
3	MW	1000	A-52B	1
4	WF	0830	B-33	1
5	WF	0830	B-13	1
6	MF	1030	C-20	5
7	MW	1400	A-09	5

EqID	Item	SeID
1	Portable Hadron Collider	NULL
2	Gravity Wave Inducer	5
3	Space Warp Drive	2
4	Worm-Hole Sequencer	NULL
5	Teleportation Disassembly Module	3
6	Teleportation Assembly Module	7
7	Coffee Maker	NULL

So far in this module, we have been using the Crow's Foot Notation, and we will continue to do so, but the Crow's Foot Notation is not the only notation used; there are many others.

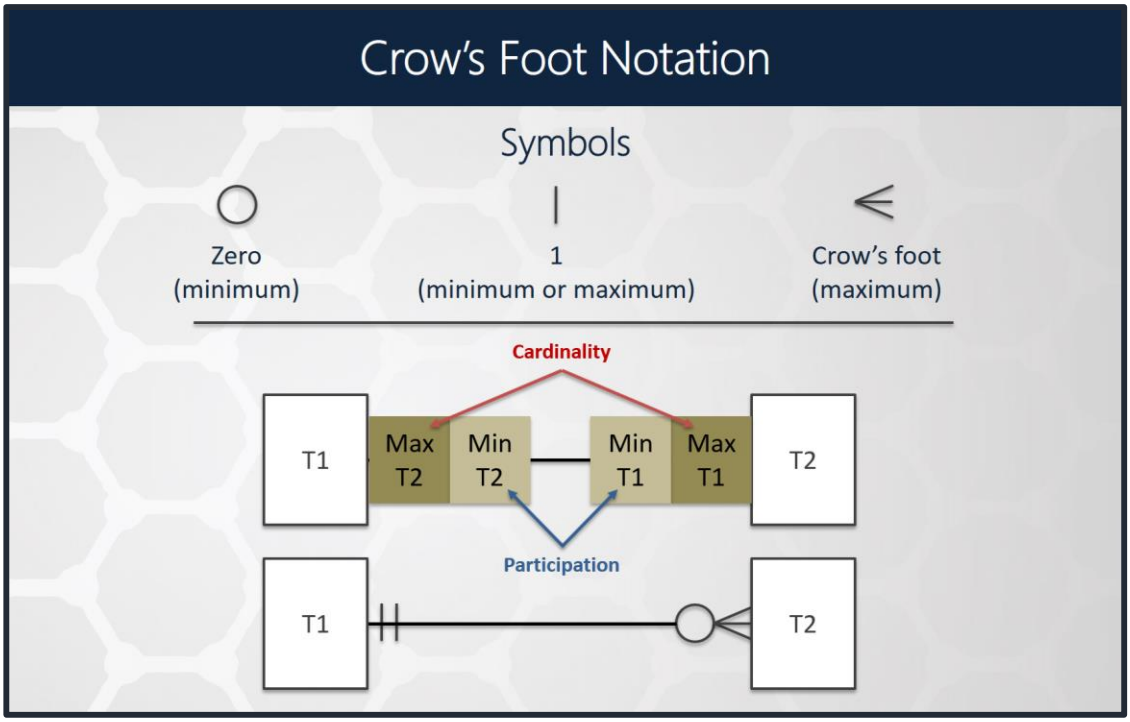
In this lesson, we will have a look at the three most common ones with special attention given to the Crow's Foot Notation. We will also discuss the Entity-Relationship Model (ERM) Notation that is very useful in certain situations.



These are only the most common of the diagraming notations you will find. Of the three shown, the Crow's Foot Notation is the most complete; it's the only one that shows table participation in addition to cardinality. The others show only the cardinality ratio.

The Relational Notation is probably the most common of the three. It shows the cardinality ratio in a simple and clean form—an arrow that points from the foreign-key table to the primary-key table.

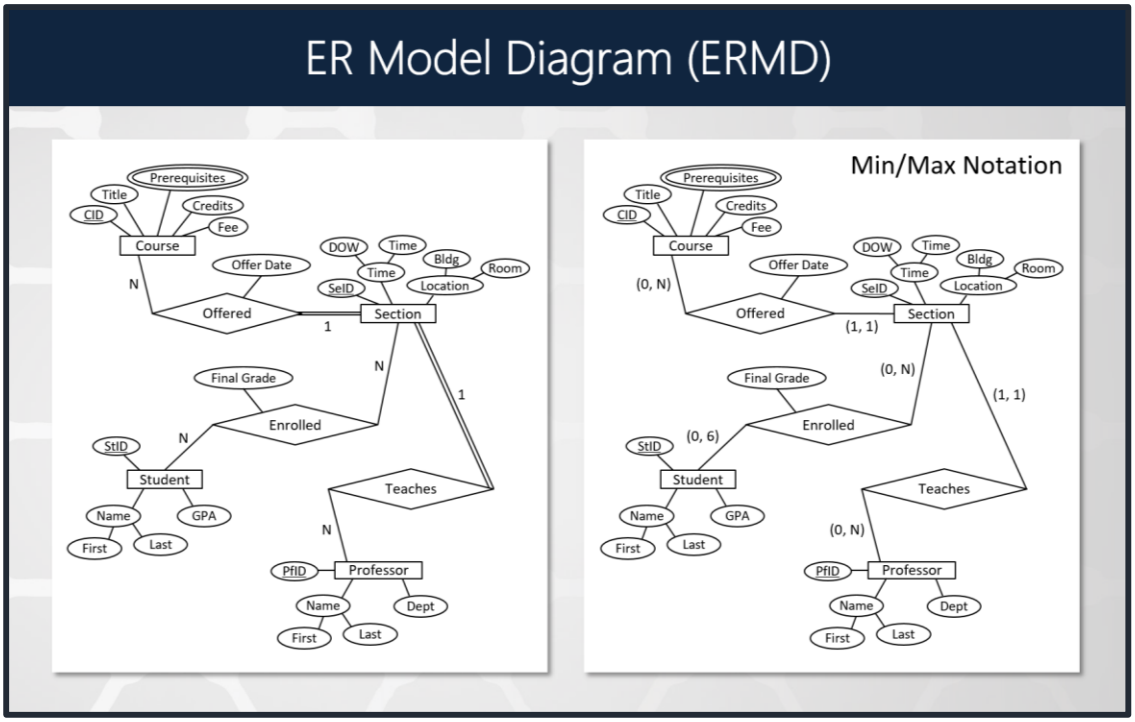
The SQL Server notation is a bit different. It uses a key for the primary key table and an infinity symbol for the foreign-key table.



As we have seen so far, the crow's foot notation is the only notation that also shows table participation in addition to cardinality.

There are three symbols used in the crow's foot notation. A zero is used only for minimums, a one is used for both maximums and minimums, and the crow's foot is used only for maximums.

The outside positions—the ones closest to the tables on the relationship line—are used for maximums and, together, denote the cardinality ratio. The inside positions—the ones farthest from the tables—are used for minimums and denote the participation of each table.



In the ERM Notation, entity-types are represented as simple rectangles, relationships are represented as diamonds with lines to the related tables, and attributes of either entity-types or relationships are represented as bubbles. Table cardinality is represented with either a 1 or N next to the table. In the diagram on the left side of the slide above, the cardinality of the Section table is 1 and of the Course table it is many. In that same diagram, table participation is represented with using a single line for partial participation or as a double line for total participation.

An alternative representation of cardinality and participation is shown in the diagram on the right side using pairs of numbers or an N. The first position in the pair represents minimums and the second one represents maximums. Cardinality and participation can be shown in greater detail this way.

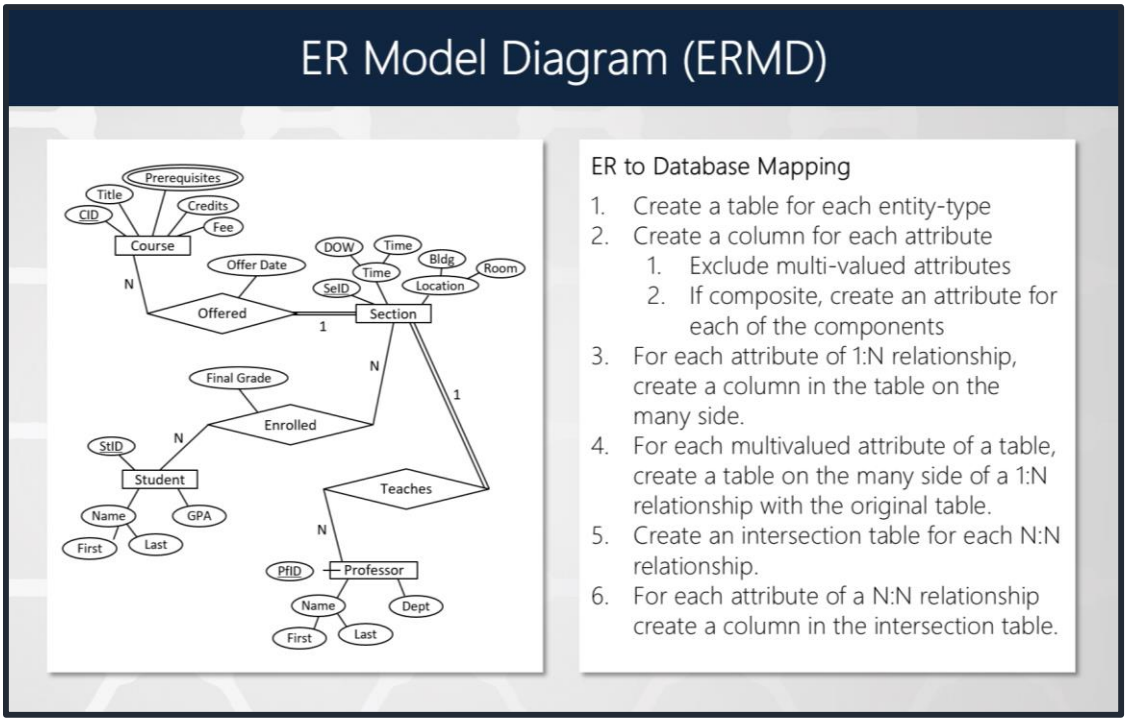
The Entity-Relationship Model Notation is popular among database designers, especially in the initial phases of the design process when the picture of the miniworld is not yet well formed in the mind of the designer and is constantly undergoing changes.

As an example, let's say the database designer discovers that there is a Course entity type with Title, Prerequisite, Credits, and Fee. Later, perhaps in the middle of an interview with one of the future users of the database, the designer realizes that one of the attributes, Prerequisite, is multivalued; each course entity can have several prerequisites. Multivalued attributes are not allowed in the database as they are a violation of First Normal Form, but rather than pull out an eraser and represent what is now the Prerequisites attribute as a table in a 1:N relationship with Course, which is the way the attribute will ultimately be represented

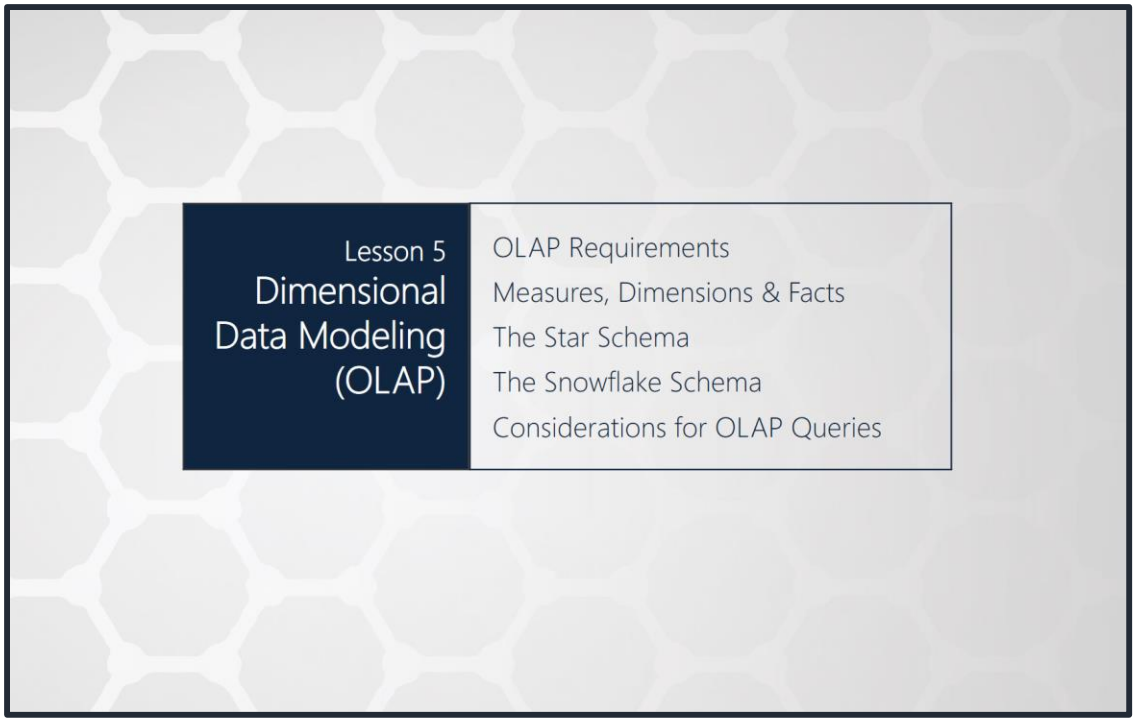
in the database, the designer can simple put another circle around it to indicate that it is multivalued, get on with the interview, and deal with it later.

Similarly, let's say the designer discovers that there is another entity-type called Section with Time and Location attributes and, while talking to other users, realizes that the attributes are composite (composed of sub-attributes). Composite attributes are also a violation of First Normal Form and are not allowed in the database. Again, rather than pull out an eraser, the designer can simple draw new bubbles on each of those two attributes to show the sub-attribute components and deal with the situation later.

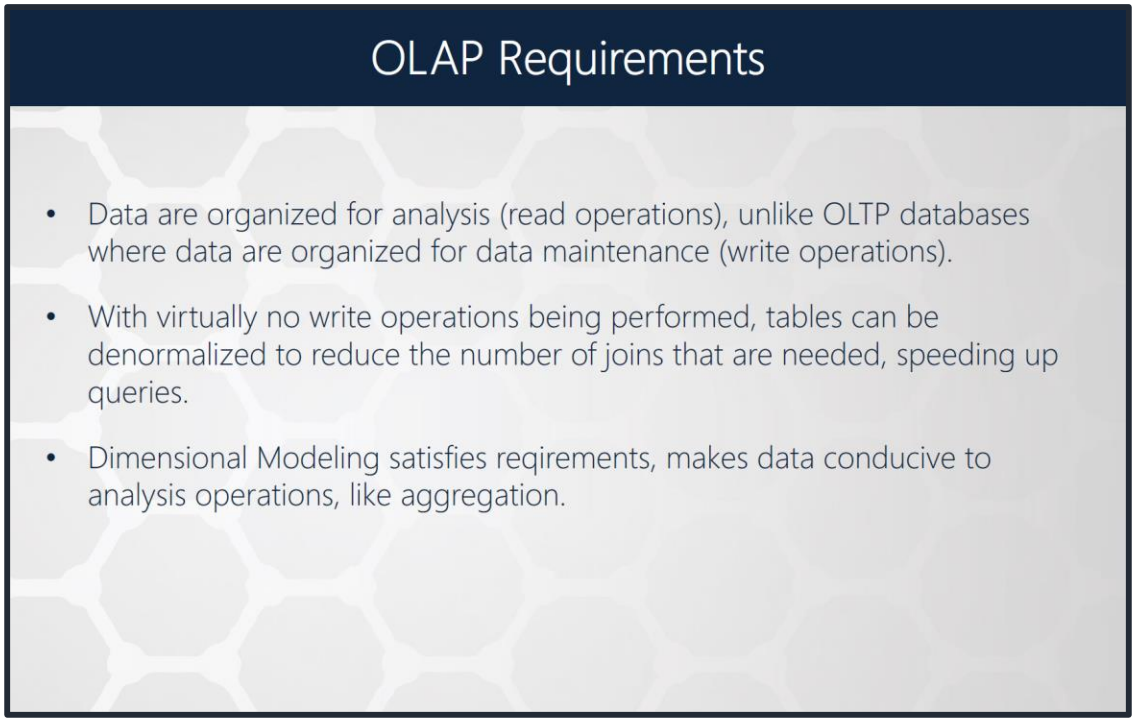
The ERM Notation is not designed to accurately represent the final form of the database, but rather to collect information that will later be refined into a final design.



When the ERM Diagram is completed, the designer must map it to the database. This is the time when multivalued and composite attributes and other aspects represented in the diagram are "resolved" and properly represented in the database.



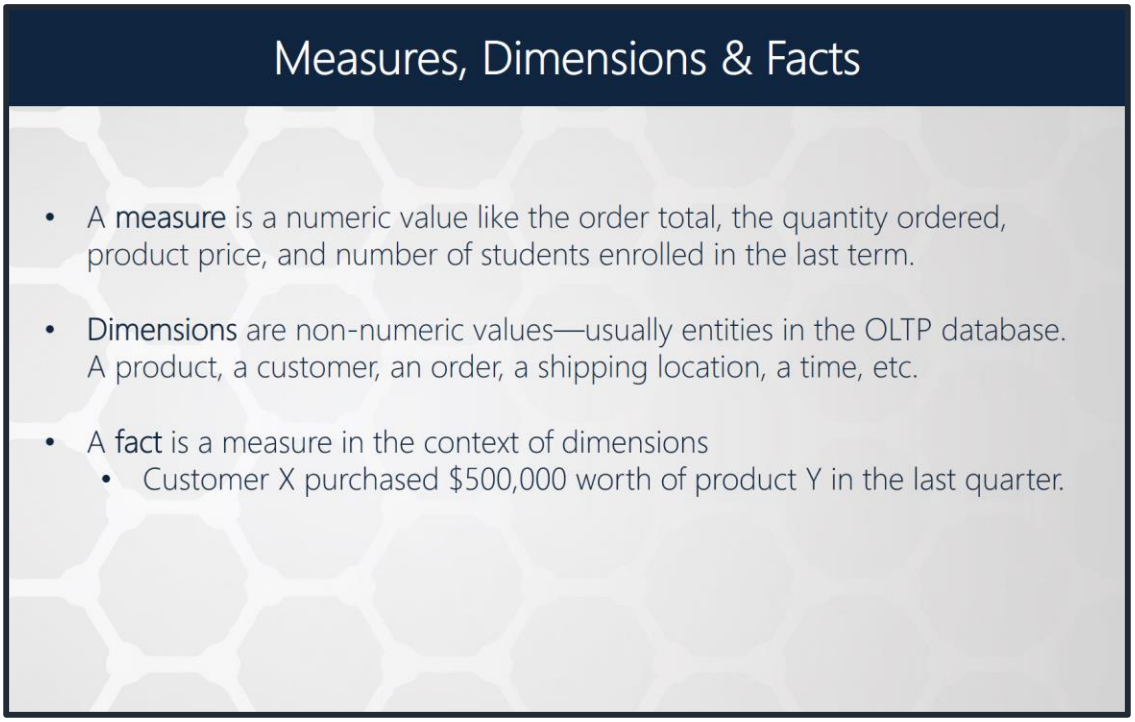
So far, we've seen how data are modeled in an OLTP environment. We now turn to OLAP environments in which data are modeled using Dimensional Modeling.



OLTP databases exist mainly for the capture and maintenance of data, so write operations are prevalent and read operations are few. By contrast, OLAP databases exist mainly for analyzing and reporting on data, so it's the other way around; read operations are prevalent and write operations virtually don't exist.

With virtually no write operations being performed, tables can be denormalized. This reduces the number of tables in the database and speeds up queries since fewer joins need to be performed.

The Dimensional Model was developed by Ralph Kimball (currently of the Kimball Group) to address the requirements of OLAP environments. It implements the star and snowflake schemas.



All data in an OLAP database fall under the categories of measures and dimensions. A measure is a numeric quantity, like the total of an order, the order quantity, the product price, and others. A dimension is a non-numeric value. Let's have a closer look at this to better understand the two and also what a fact is.

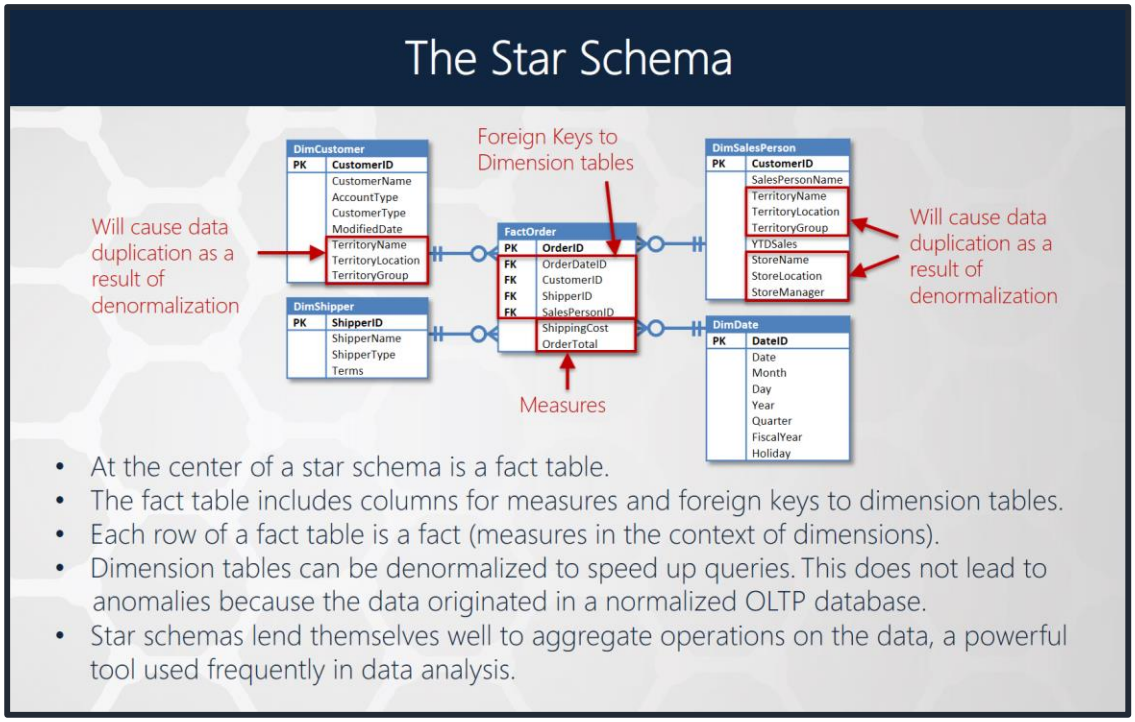
Let's say you just joined a meeting in progress in Conference Room B. Shortly after you sit down, someone opens the door to the conference room, looks at one of the meeting participants and says, "$500,000," then closes the door and disappears. Understanding nothing of what just happened, you look around the room confused and notice that all the meeting participants but you seem to understand exactly what just happened. One of them, Susan, noticing your confusion, turns to you and explains, "Mary had asked Bob to find out Customer X's total sales for Product Y in the last quarter."

"Oh," you say. "Thanks."

How did Susan give meaning to the $500,000 that had previously had none? She put the $500,000 measure in the context of dimensions and produced a fact. Susan associated the measure to a customer dimension (Customer X), a product dimension (Product Y), and a time dimension (last quarter) and, when she did, the result was a *fact*.

Dimensions give measures meaning by putting them into context. When a measure is put into the context of one or more dimensions, the result is a *fact*. It is a fact that Customer X purchased $500,000 worth of Product Y in the last quarter.

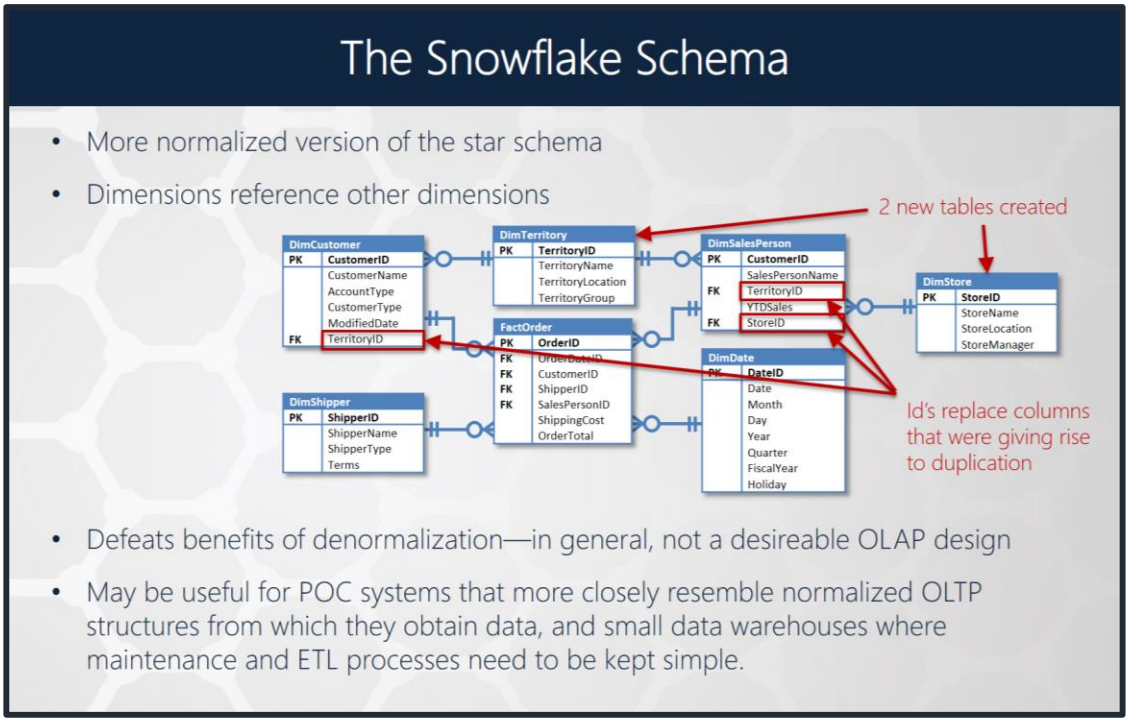
In the dimensional model, data are represented as facts that are quantified by measures and qualified by dimensions.



How is all this represented in the dimensional model? We start with a *fact table*. A fact table has two types of columns: measures and dimensions. Done. Every row of a fact table puts measures in the context of the dimensions it relates to and is thus a fact.

If additional attributes about dimensions beyond just their identification is needed—and it almost always is; customers have names, contact information, territories, credit limits, and so on—a dimension table is created for that dimension and a corresponding foreign key is added to the fact table. Queries can then join fact tables to dimension tables and be able to use any one of their attributes for data aggregation and other nalytical operations on the data.

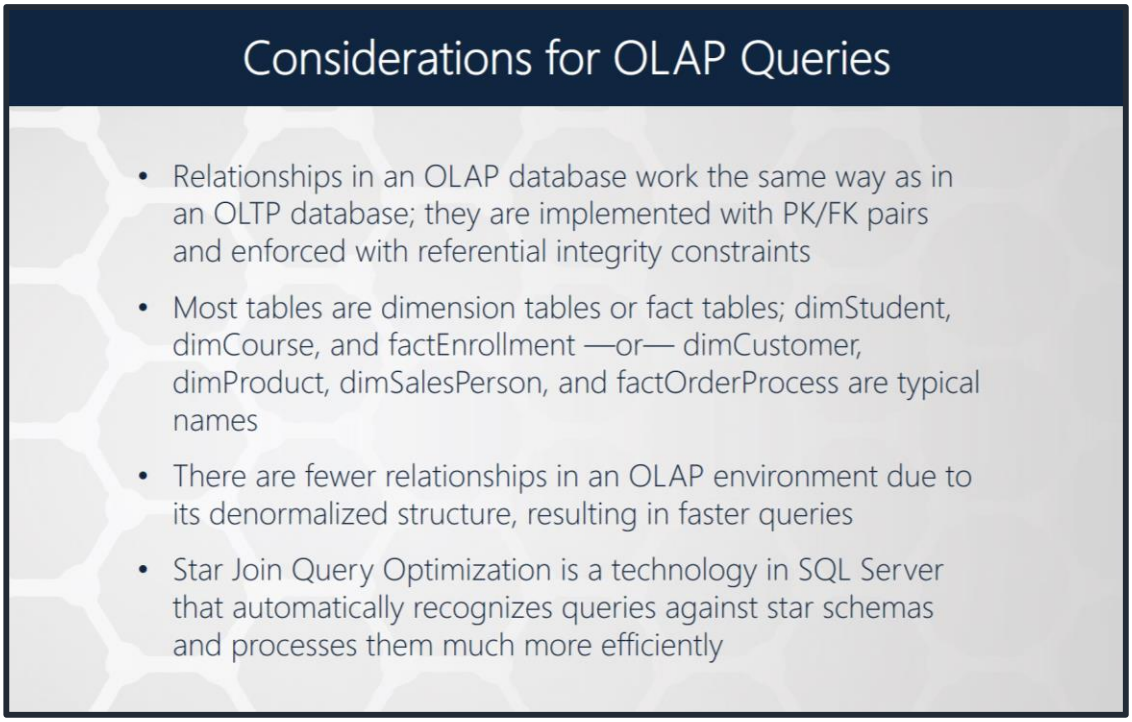
The only relationships in a star schema are the ones between the fact tables and their dimensions; there are no relationships among dimensions. If a dimension is related to others, then one of them will be stored together with the other. This leads to data duplication in OLAP databases, though not to anomalies since the data originated from one or more OLTP environments.



The snowflake schema contains relationships among dimensions. This reduces data duplication but requires more joins in queries.

As a more normalized structure, the snowflake schema represents the data in formats that are closer to their formats in the OLTP environment. This can make proof of concept (POC) systems easier to work with. Once the concept is proven, the snowflake schema can be replaced by a star schema during implementation.

In general, a snowflake schema is not the best choice for OLAP environments because it defeats the significant benefits of denormalization.



OLAP databases stored in SQL Server are databases like any other SQL Server database and are queried using SQL in the same way. Almost every table in an OLAP database is a dimension table or a fact table. Tables commonly use the prefixes "dim" and "fact". Names like dimCourse, dimStudent, factEnrollment or dimCustomer, dimProduct, dimSalesOrder, factOrderProcess are typical.

The OLAP environment has a number of traits that speed up queries. Because it's a denormalized environment, there are fewer tables to join in queries reducing processing and data overhead. Materialized views reduce execution time. Data partitioning and compression reduce the amount of data a query has to sift through, and there many others, including SJQO technology.

Star Join Query Optimization is capable of recognizing that a query is running against a star schema and uses a system of pre-setup hash tables to greatly speed up query execution. SJQO does not need to be turned on, set up, or anything like that. It's built in to SQL Server.

Module 1 Summary and Review

- A database is a model of a miniworld that organizes data in a way that is conducive to information retrieval.
- Information is derived from the data using queries.
- Relational databases implement the relational model to describe data in an abstract, conceptual manner. The relational model's strong mathematical foundation gives it a lot of flexibility and power.
- There are two general types of relational databases: OLTP where writing operations prevail and OLAP where read operations prevail. Writing and reading operations do not coexist well.
- It is necessary to understand how relational databases organize data if we are to write effective queries that give us the information we want.
- In an OLTP environment, we identified a number of building blocks: *Entities* are things that exist in the miniworld and that have *attributes* of interest. Entities of the same type can be grouped into an *entity-type*. Entities of the same entity-type have the same attributes.
- For each entity-type in the miniworld, a table is created in the database. The columns of the table represent the attributes of the entity-type, and the rows represent the individual entities.
- As a rule, each table represents one and only one entity-type.
- A primary key is a unique identifier in a table that is used to identify individual entities to other entities in other tables. A table can have only one primary key.
- The best primary key values are fact-less, ever-increasing integers.
- Entities in different entity-types can be related to each other. Relationships have 3 properties.
    - The name of a relationship should be descriptive of the type of interaction between the two related tables that the relationship represents.
    - Relationship type describes the ratio between the maximum number of relationships that a row in one table can have with another table and that same maximum from the perspective of the other table. To determine the type of relationship between two tables, A & B, we answer two questions

        1. For each row in Table A, how many rows can there be in Table B, one or many?

        and

2. For each row in Table B, how many rows can there be in Table A, one or many?

   If the answer to both questions is one, then the relationship is 1:1, if the answer to one question is many and to the other it is one, then the relationship is 1:N, and if the answer to both questions is many, then the relationship is N:M.

- Table participation describes the minimum number of relationships that a row in one of the tables must have with the other table. It is either zero or one. If a table has a minimum of zero, then its participation in the relationship is said to be *partial*, and if it's one, then it is said to be *total*.

- All relationships are implemented using primary key/foreign key pairs:
  - In a 1:N relationship, a foreign key is created in the entity-type on the many side of the relationship that references the primary key in the entity-type on the one side of the relationship.
  - In a N:M relationship, an intersection table containing 2 foreign keys is created. One foreign key references the primary key of one of the original tables, and the other foreign key references the primary key of the other original table.
  - In a 1:1 relationship, the primary key in the table with total participation doubles-up as a foreign key that references the primary key in the other table. If there is no table with total participation, then one of the tables is chosen arbitrarily for the foreign key.
- There are many diagramming notations to represent relational database schemas. We used the Crow's Foot notation because it also includes table participation. The symbols closest to the tables along the relationship lines represent table cardinality and the symbols farthest from the tables represent table participation.
- OLAP databases also represent data in tables related using PK/FK pairs, though tables in an OLAP database are either fact tables or dimension tables. A fact is a measure in the context of one or more dimensions.
- OLAP databases are optimized for read operations and so queries run very efficiently in them.