رواد مصر الرقمية

وزارة الاتصالات
وتكنولوجيا المعلومات

Databases and SQL for Data Science with Python

03/08/2024

Database

Organized collection of inter-related data that models some aspect of the real-world.

Databases are the core component of most computer applications.

Database Example

Create a database that models a digital music store to keep track of artists and albums.

Things we need for our store:

→ Information about Artists

→ What Albums those Artists released

Flat File Strawman

Store our database as comma-separated value (CSV) files that we manage ourselves in our application code.

→ Use a separate file per entity.

→ The application must parse the files each time they want to read/update records.

Artist(name, year, country)

```
"Wu-Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"GZA",1990,"USA"
```

Album(name, artist, year)

```
"Enter the Wu-Tang","Wu-Tang Clan",1993
"St.Ides Mix Tape","Wu-Tang Clan",1994
"Liquid Swords","GZA",1990
```

Flat File Strawman

Example: Get the year that GZA went solo.

```
Artist(name, year, country)
"Wu-Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"GZA",1990,"USA"
```

```
for line in file.readlines():
  record = parse(line)
  if record[0] == "GZA":
    print(int(record[1]))
```

Flat Files: Data Integrity

How do we ensure that the artist is the same for each album entry?

What if somebody overwrites the album year with an invalid string?

What if there are multiple artists on an album?

What happens if we delete an artist that has albums?

Flat Files: Implementation

How do you find a particular record?

What if we now want to create a new application that uses the same database?
What if that application is running on a different machine?

What if two threads try to write to the same file at the same time?

Flat Files: Durability

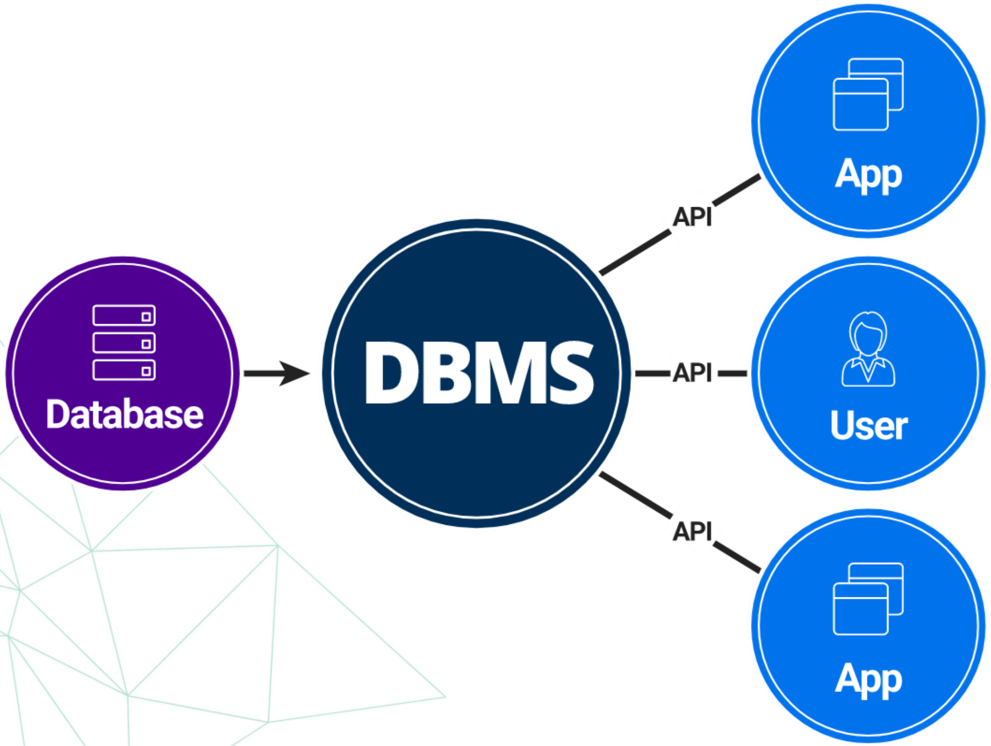
What if the machine crashes while our program is updating a record?

What if we want to replicate the database on multiple machines for high availability?

Database Management System

A database management system (DBMS) is software that allows applications to store and analyze information in a database.

A general-purpose DBMS supports the definition, creation, querying, update, and administration of databases in accordance with some data model.



Data Models

A data model is an abstract model that organizes data elements and standardizes how they relate to one another and to the properties of real-world entities. It provides a high-level conceptual understanding of the data.

Purpose:

- To define the structure, relationships, and constraints of the data.
- To serve as a blueprint for creating and managing a database.
- To ensure consistency and data integrity across the system.

Components:

- Entities (e.g., Customer, Order)
- Attributes (e.g., Customer Name, Order Date)
- Relationships (e.g., A Customer can place multiple Orders)
- Constraints (e.g., An Order must have a Customer)

Data Models

A schema is a formal description of how the data is organized within a database. It defines the structure and organization of the database and its objects.

Purpose:

- To define the actual physical implementation of the data model within a specific database management system (DBMS).
- To provide a detailed blueprint for the database's structure and organization.

Components:

- Tables (e.g., Customers, Orders)
- Columns/Fields (e.g., CustomerID, OrderDate)
- Data Types (e.g., VARCHAR, INT, DATE)
- Indexes
- Constraints (e.g., Primary Key, Foreign Key)
- …

Data Models

Relational ← Most DBMSs

Key/Value
Graph
Document / XML / Object
Wide-Column / Column-family
← NoSQL

Array / Matrix / Vectors ← Machine Learning

Hierarchical
Network
Multi-Value
← Obsolete / Legacy / Rare

Data Models

Relational ← This Course

Key/Value
Graph
Document / XML / Object
Wide-Column / Column-family

Array / Matrix / Vectors

Hierarchical
Network
Multi-Value

Relational Model

The relational model defines a database abstraction based on relations to avoid maintenance overhead.

Key tenets:

→ Store database in simple data structures (relations).

→ Physical storage left up to the DBMS implementation.

→ Access data through high-level language, DBMS figures out best execution strategy.

Relational Model

Structure:
The definition of the database's relations and their contents.

Integrity:
Ensure the database's contents satisfy constraints.

Manipulation:
Programming interface for accessing and modifying a database's contents.

Relational Model

A relation is an unordered set that contain the relationship of attributes that represent entities.

A tuple is a set of attribute values (also known as its domain) in the relation.

→ Values are (normally) atomic/scalar.
→ The special value NULL is a member of every domain (if allowed).

n-ary Relation

=

Table with n columns

`Artist(name, year, country)`

name	year	country
Wu-Tang Clan	1992	USA
Notorious BIG	1992	USA
GZA	1990	USA

Relational Model: Primary Keys

A relation's primary key uniquely identifies a single tuple.

Some DBMSs automatically create an internal primary key if a table does not define one.

DBMS can auto-generation unique primary keys via an identity column:

→ IDENTITY (SQL Standard?)
→ SEQUENCE (PostgreSQL / Oracle)
→ AUTO_INCREMENT (MySQL)

Artist(id, name, year, country)

id	name	year	country
101	Wu-Tang Clan	1992	USA
102	Notorious BIG	1992	USA
103	GZA	1990	USA

Relational Model: Foreign Keys

A <u>foreign key</u> specifies that an attribute from one relation maps to a tuple in another relation.

ArtistAlbum(<u>artist_id</u>, <u>album_id</u>)

artist_id	album_id
101	11
101	22
103	22
102	22

Artist(<u>id</u>, name, year, country)

id	name	year	country
101	Wu-Tang Clan	1992	USA
102	Notorious BIG	1992	USA
103	GZA	1990	USA

Album(<u>id</u>, name, ~~artists~~, year)

id	name	artists	year
11	Enter the Wu-Tang		1993
22	St.Ides Mix Tape	??	1994
33	Liquid Swords		1995

Relational Model: Constraints

User-defined conditions that must hold for any instance of the database.

→ Can validate data within a single tuple or across entire relation(s).
→ DBMS prevents modifications that violate any constraint.

Artist(<u>id</u>, name, year, country)

id	name	year	country
101	Wu-Tang Clan	1992	USA
102	Notorious BIG	1992	USA
103	GZA	1990	USA

Relational Model: Queries

The relational model is independent of any query language implementation.

SQL is the *de facto* standard (many dialects).

```
for line in file.readlines():
  record = parse(line)
  if record[0] == "GZA":
    print(int(record[1]))
```

```
SELECT year FROM artists
 WHERE name = 'GZA';
```

Data Models

Relational

Key/Value
Graph
Document / XML / Object ⟵ Leading Alternative
Wide-Column / Column-family

Array / Matrix / Vectors ⟵ Current Hotness

Hierarchical
Network
Multi-Value

Document Data Model

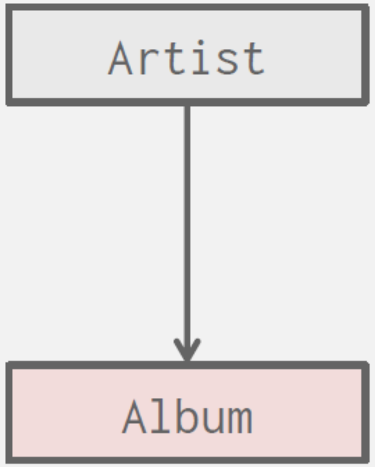
A collection of record documents containing a hierarchy of named field/value pairs.

→ A field's value can either a scalar type, an array of values, or another document.

→ Modern implementations use JSON. Older systems use XML or custom object representations.

Avoid "relational-object impedance mismatch" by tightly coupling objects and database.

Document Data Model



Application Code

```
class Artist {
   int id;
   String name;
   int year;
   Album albums[];
}
class Album {
  int id;
  String name;
  int year;
}
```

```
{
 "name": "GZA",
 "year": 1990,
 "albums": [
   {
     "name": "Liquid Swords",
     "year": 1995
   },
   {
     "name": "Beneath the Surface",
     "year": 1999
  }
 ]
}
```

Vector Data Model

One-dimensional arrays used for nearest-neighbor search (exact or approximate).

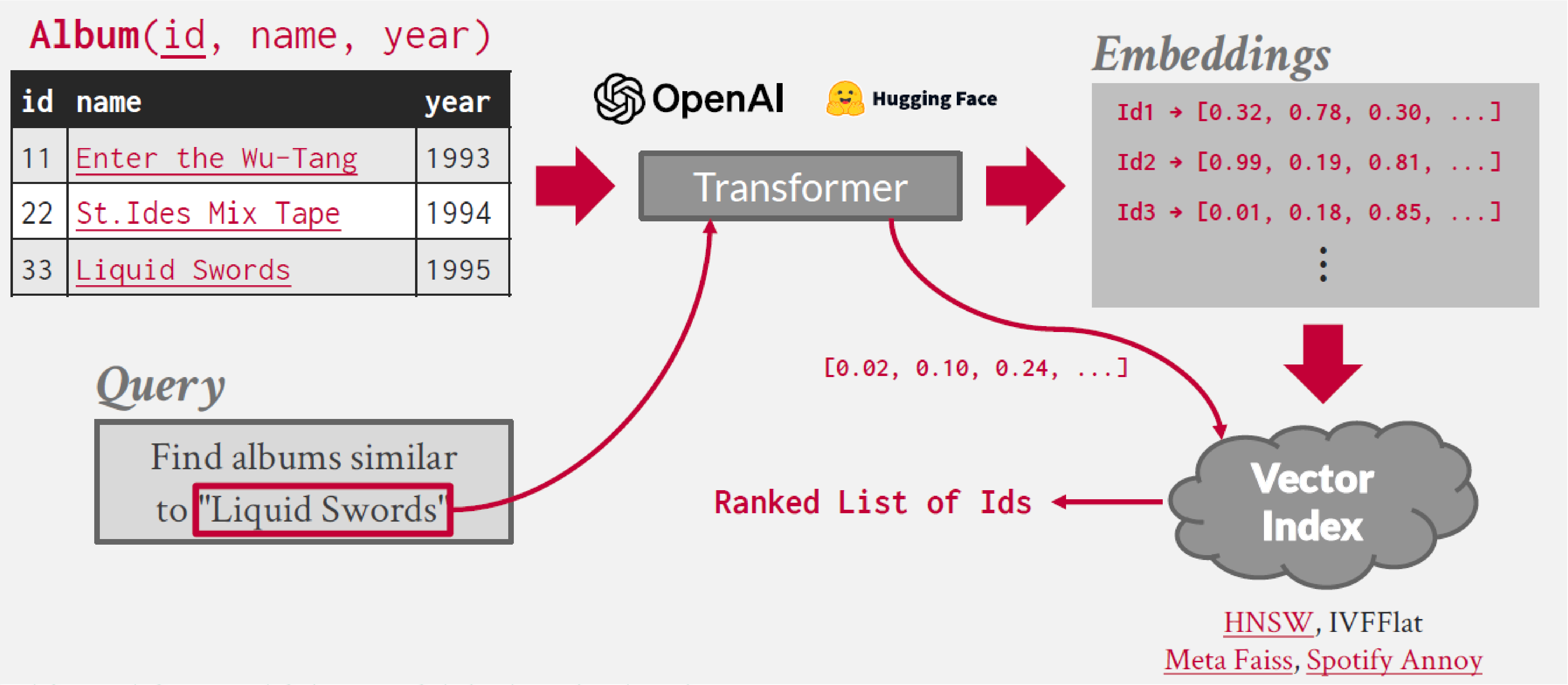
→ Used for semantic search on embeddings generated by ML-trained transformer models (think ChatGPT).

→ Native integration with modern ML tools and APIs (e.g., LangChain, OpenAI).

At their core, these systems use specialized indexes to perform NN searches quickly.

Pinecone Weaviate milvus qdrant
marqo LanceDB featureform

Vector Data Model



Relational Databases

Relational Databases

A relational database is a way to organize data in a structured format. Imagine it as a collection of tables, similar to spreadsheets. Each table has rows (records) and columns (fields).

The power of relational databases lies in how these tables can be related to each other based on shared information, creating a network of interconnected data.

Key characteristics:

Tables: Store data in a structured format with rows and columns.

Relationships: Connect tables based on shared information, allowing for complex queries.

Data integrity: Ensures data consistency and accuracy through constraints and rules.

Efficiency: Optimized for storing and retrieving large amounts of data.

Relational Databases

For example:

Column, Attribute, or Field

Row or Record

Patient Name	Patient Birthdate	Patient Phone Number
Diane Hyson	3/4/1970	(540) 555-1212
Leon Stevens	11/10/1952	(703) 555-1234

Relational Databases

To illustrate an example of a relationship between database tables:

Patients

Patient Name	Patient Birthdate	Patient Phone Number
Diane Hyson	3/4/1970	(540) 555-1212
Leon Stevens	11/10/1952	(703) 555-1234

Appointments

Patient Name	Appointment Time	Appointment Reason	Doctor Name
Diane Hyson	2/28/2020 2:30pm	Annual Check-Up	Dr. Urena
Leon Stevens	3/2/2020 10:00am	Treatment	Dr. Hammad
Leon Stevens	3/9/2020 10:00am	Follow-Up	Dr. Hammad

The connection between these two tables could be the patient's name. (In reality, a unique identifier would be assigned to each patient, since two people can have the same name, but for this illustration, the name will suffice.)

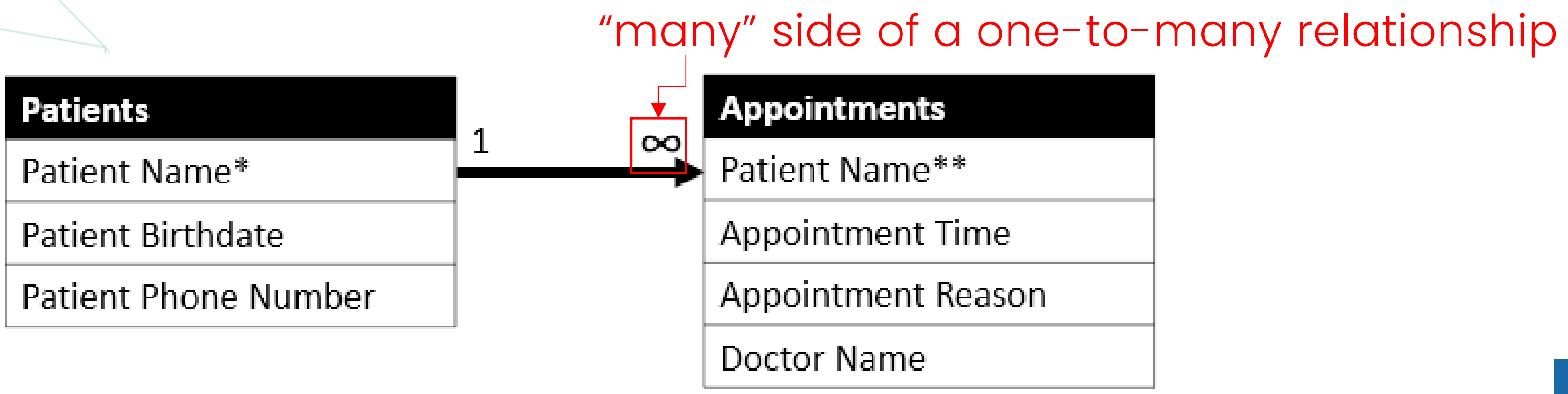
In order to create a report of every patient who has an appointment scheduled in the next week along with their contact information, there would have to be an established connection between the patient directory table and the appointment table, enabling someone to pull data from both tables simultaneously.

Relational Databases

The relationship between the entities just described is called a *<u>one-to-many</u>* relationship.

Each patient only appears in the patient directory table one time but can have many appointments in the related appointment-tracking table. Each appointment only has one patient's name associated with it.

Database relationships like this one are depicted in what's called an *<u>entity-relationship diagram (ERD).</u>* The ERD for these two tables is:



Relational Databases: *Primary vs. Foreign*

A *primary key* uniquely identifies a row in a table using one or more columns, where the combination of values must be unique and non-null. It can be based on unique data values, like a Student ID, or generated by the database, such as an auto-incrementing integer.

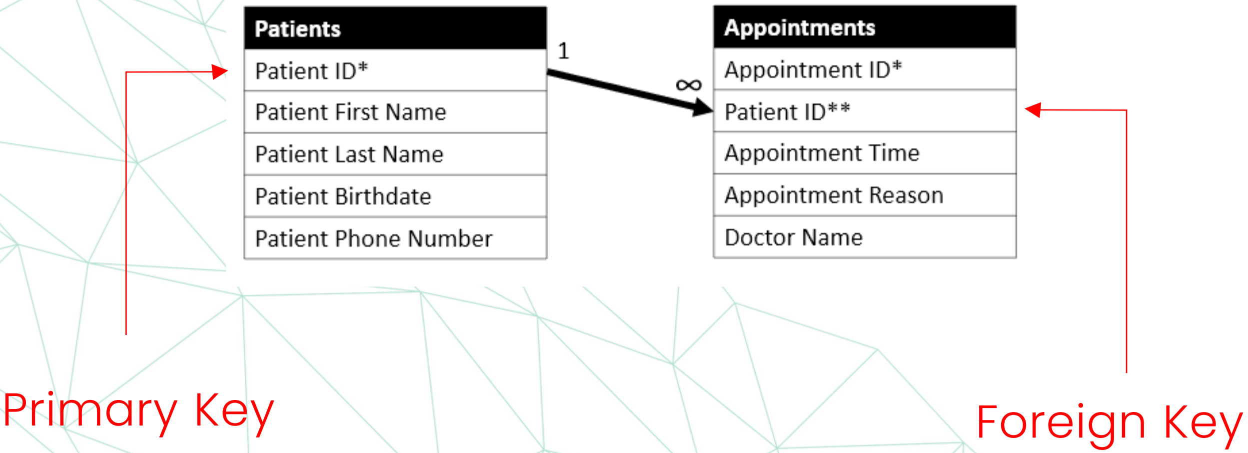
Primary keys can be used to identify related records in other tables, where they are referred to as *foreign keys*.

Using Patient Name as a primary key is a poor choice because names are not unique. A common practice is to create an *auto-incrementing integer* field to serve as a unique identifier.

This avoids relying on potentially sensitive or unavailable information, such as Social Security numbers.

Relational Databases: *Primary vs. Foreign*

The doctor's office database assigns auto-incrementing integers as primary keys for patient and appointment records. The appointment-tracking table uses the generated Patient ID to link appointments to patients, eliminating the need to store patient names in the Appointments table.



Patients

Patient ID	Patient First Name	Patient Last Name	Patient Birthdate	Patient Phone Number
1	Diane	Hyson	3/4/1970	(540) 555-1212
2	Leon	Stevens	11/10/1952	(703) 555-1234

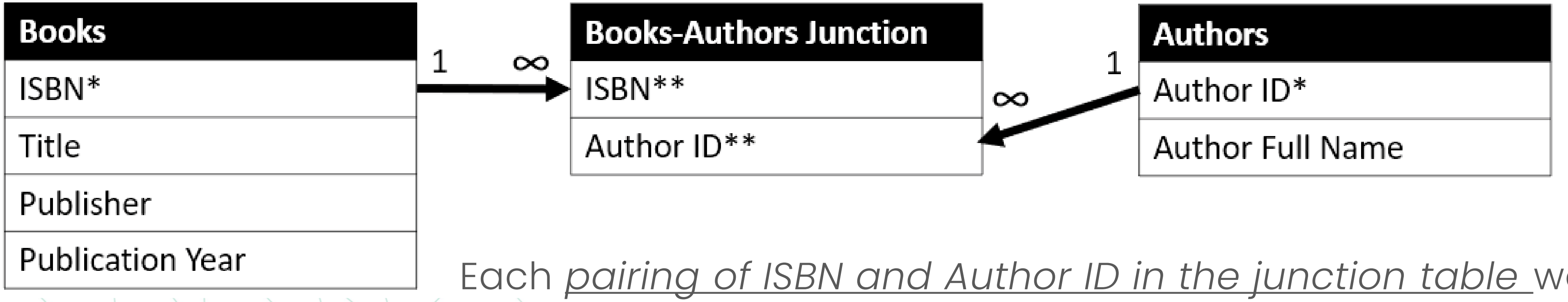
Appointments

Appointment ID	Patient ID	Appointment Time	Appointment Reason	Doctor Name
100	1	2/28/2020 2:30pm	Annual Check-Up	Dr. Urena
101	2	3/2/2020 10:00am	Treatment	Dr. Hammad
102	2	3/9/2020 10:00am	Follow-Up	Dr. Hammad

Relational Databases

A many-to-many relationship in RDBMSs allows records on each side to connect to multiple records on the other.
For example, a Books and Authors table has a *many-to-many* relationship: each author can write multiple books, and each book can have multiple authors.

This relationship requires a junction table to capture the pairs of related rows.



Each *pairing of ISBN and Author ID in the junction table* would be unique, so the pair of fields can be considered a multi-column primary key in the Books-Authors Junction table.

Relational Databases

By using a junction table to handle relationships, we avoid storing multiple rows per book or multiple authors per book directly in the Books table. This reduces redundant data and clarifies entity relationships.

his process, known as database normalization, ensures each author's name is stored once, regardless of how many books they've written. Similarly, a patient's phone number is stored only in the patient directory, not repeatedly in the Appointments table, because it’s already stored in the related patient directory table and can be found by connecting the two tables via the Patient ID using SQL JOINs.

Normalization reduces storage needs and simplifies data updates.

Structured Query Language (SQL)

Structured Query Language (SQL)

SQL (Structured Query Language) is the standard language used to interact with relational databases.

Early experiences with relational database design often begin with MS Access, where basic SQL concepts are learned.

These concepts are applicable across a career and with various RDBMSs such as MS SQL Server, Oracle Database, MySQL, and Amazon Redshift. While syntax may vary, the core concepts remain consistent.

SQL-style RDBMSs were first developed in the 1970s, and the basic database design concepts have stood the test of time; many of the database systems that originated then are still in use today. The longevity of these tools is another reason that SQL is so ubiquitous and so valuable to learn.

Structured Query Language (SQL)

As a professional who works with data, you will likely encounter several of the following popular Relational Database Management Systems:

- $\boxed{\text{MySQL}}$ ← This Course
- Oracle
- MS SQL Server
- PostgreSQL
- Amazon Redshift
- IBM DB2
- MS Access
- SQLite
- Snowflake

Relational Languages

In SQL, we divide the language into several sublanguages to handle different aspects of database management. Here's a breakdown:

1. **Data Definition Language (DDL):** DDL is responsible for defining the structure or schema of the database.

   CREATE | ALTER | DROP | TRUNCATE

2. **Data Manipulation Language (DML):** DML deals with manipulating data itself.

   SELECT | INSERT | UPDATE | DELETE

3. **Data Control Language (DCL):** DCL handles administrative tasks related to controlling the database.

   GRANT | REVOKE | DENY

4. **Transaction Control Language (TCL):** TCL Deals with the transactions within the database.

   COMMIT | ROLLBACK | SAVEPOINT

Introduction to the Farmer’s Market Database

Farmer’s Market Database

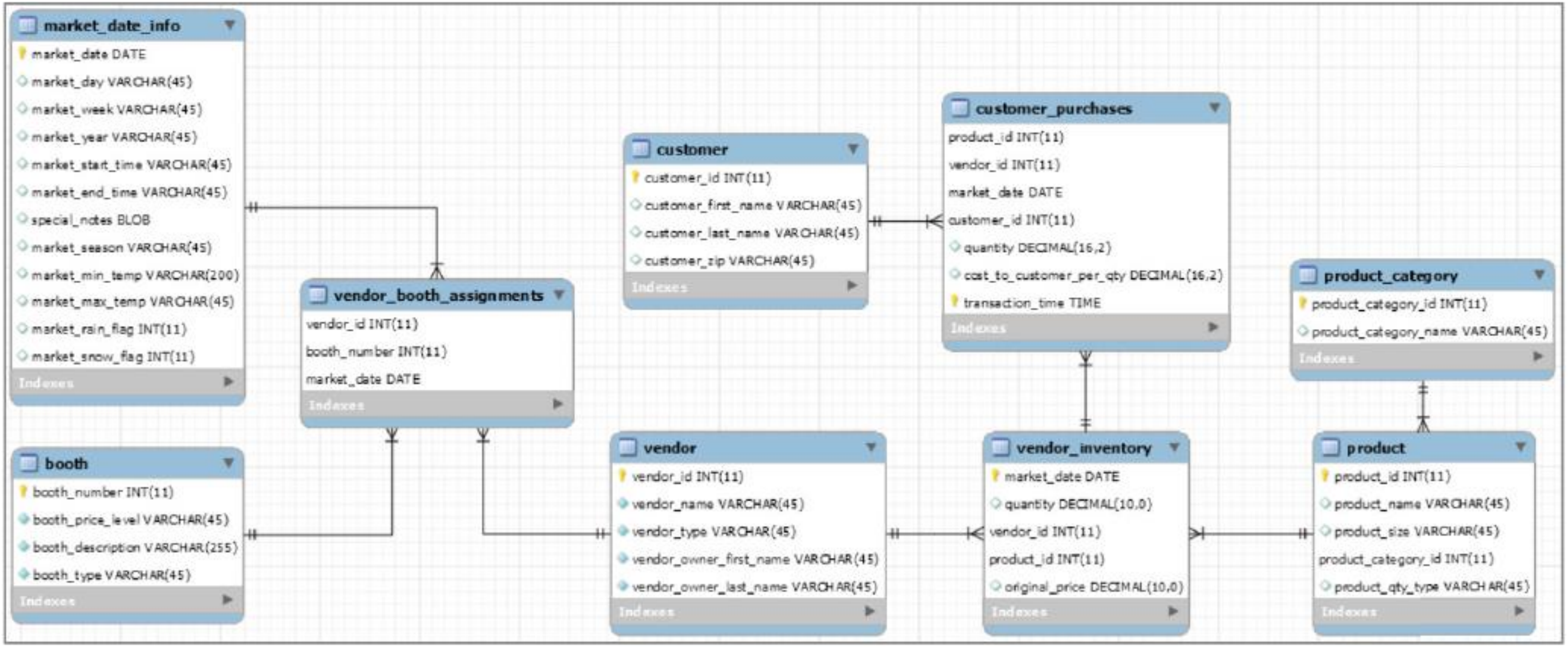
The MySQL database we’ll be using for example queries throughout much of this module serves as a tracking system for vendors, products, customers, and sales at a fictional farmer’s market.

This relational database contains information about each day the farmer’s market is open, such as the date, the hours, the day of the week, and the weather.

There is data about each vendor, including their booth assignments, products, and prices.

Farmer’s Market Database

The ERD for the entire Farmer’s Market database.



SQL - Retrieving Data

The **SELECT** Statement

A SELECT statement retrieves data from a database and, when combined with other SQL keywords, can view columns, combine tables, filter results, perform calculations, and more.

SQL SELECT queries follow this basic syntax, though most of the clauses are optional:

```
SELECT [columns to return]
FROM [schema.table]
WHERE [conditional filter statements]
GROUP BY [columns to group on]
HAVING [conditional filter statements that are run after grouping]
ORDER BY [columns to sort on]
```

Generally Required

The **SELECT** Statement

Selecting Columns and Limiting the Number of Rows Returned, The simplest SELECT statement is:

```
SELECT * FROM [schema.table]
```

For example,

```
SELECT * FROM farmers_market.product
```

Can be read as "Select everything from the product table in the farmers_market schema." The asterisk really represents "all columns".

There is a LIMIT clause which sets the maximum number of rows that are returned.

The **SELECT** Statement

By using LIMIT, you can get a preview of your current query's results without having to wait for all of the results to be compiled. For example,

```
SELECT *
FROM farmers_market.product
LIMIT 5
```

Multiple lines for readability

returns all columns for the first 5 rows of the product table:

product_id	product_name	product_size	product_category_id	product_qty_type
1	Habanero Peppers - Organic	medium	1	lbs
2	Jalapeno Peppers - Organic	small	1	lbs
3	Poblano Peppers - Organic	large	1	unit
4	Banana Peppers - Jar	8 oz	3	unit
5	Whole Wheat Bread	1.5 lbs	3	unit

The **SELECT** Statement

To specify which columns you want returned, list the column names immediately after SELECT, separated by commas, instead of using the asterisk.

```
SELECT product_id, product_name
FROM farmers_market.product
LIMIT 5
```

This gives,

product_id	product_name
1	Habanero Peppers - Organic
2	Jalapeno Peppers - Organic
3	Poblano Peppers - Organic
4	Banana Peppers - Jar
5	Whole Wheat Bread

The **SELECT** Statement

TIP Even if you want all columns returned, it's good practice to list the names of the columns instead of using the asterisk, especially if the query will be used as part of a data pipeline (if the query is automatically run nightly and the results are used as an input into the next step of a series of code functions without human review, for example). This is because returning "all" columns may result in a different output if the underlying table is modified, such as when a new column is added, or columns appear in a different order, which could break your automated data pipeline.

The **SELECT** Statement

The following query lists five rows of farmer's market vendor booth assignments, displaying the market date, vendor ID, and booth number from the vendor_booth_assignments table

```
SELECT market_date, vendor_id, booth_number
FROM farmers_market.vendor_booth_assignments
LIMIT 5
```

market_date	vendor_id	booth_number
2019-03-13	4	2
2019-03-09	3	1
2019-03-02	4	7
2019-03-02	1	2
2019-03-13	8	6

But this output would make a lot more sense if it were sorted by the market date, right?

The **ORDER BY** Clause: Sorting Results

The ORDER BY clause sorts output rows by specifying columns and their sort order—ascending (ASC) or descending (DESC). ASC sorts text alphabetically and numbers from low to high, while DESC reverses the order. In MySQL, NULL values appear first in ascending order.

The following query sorts the results by product name, even though the product ID is listed first in the output:

```
SELECT product_id, product_name
FROM farmers_market.product
ORDER BY product_name
LIMIT 5
```

product_id	product_name
7	Apple Pie
4	Banana Peppers - Jar
8	Cherry Pie
6	Cut Zinnias Bouquet
10	Eggs

The **ORDER BY** Clause: Sorting Results

And the following modification to the ORDER BY clause changes the query to now sort the results by product ID, highest to lowest,

```
SELECT product_id, product_name
FROM farmers_market.product
ORDER BY product_id DESC
LIMIT 5
```

product_id	product_name
11	Pork Chops
10	Eggs
9	Sweet Potatoes
8	Cherry Pie
7	Apple Pie

Note that the rows returned display a different set of products than we saw in the previous query. That’s because the ORDER BY clause is executed before the LIMIT is imposed.

The **ORDER BY** Clause: Sorting Results

We can sort by multiple columns by adding an ORDER BY line, specifying that we want it to sort the output first by market date, then by vendor ID.

```
SELECT market_date, vendor_id, booth_number
FROM farmers_market.vendor_booth_assignments
ORDER BY market_date, vendor_id
LIMIT 5
```

market_date	vendor_id	booth_number
2019-03-02	1	2
2019-03-02	3	1
2019-03-02	4	7
2019-03-02	7	11
2019-03-02	8	6

Simple Inline Calculations

Let's say we wanted to do a calculation using the data in two different columns in each row. In the customer_purchases table, we have a quantity column and a cost_to_customer_per_qty column, so we can multiply those to get a price.

```
SELECT
     market_date,
     customer_id,
     vendor_id,
     quantity,
     cost_to_customer_per_qty
FROM farmers_market.customer_purchases
LIMIT 10
```

market_date	customer_id	vendor_id	quantity	cost_to_customer_per_qty
2019-03-02	4	8	2.00	4.00
2019-03-02	10	8	1.00	4.00
2019-03-09	12	8	1.00	4.00
2019-03-09	5	9	1.00	16.00
2019-03-09	1	9	1.00	18.00
2019-03-02	2	4	4.60	2.00
2019-03-02	3	4	8.40	2.00
2019-03-02	4	4	1.40	2.00
2019-03-09	4	4	9.90	2.00
2019-03-02	1	1	1.00	5.50

Simple Inline Calculations

The following query demonstrates how the values in two different columns can be multiplied by one another by putting an asterisk between them.

```
SELECT
     market_date,
     customer_id,
     vendor_id,
     quantity,
     cost_to_customer_per_qty,
     quantity * cost_to_customer_per_qty
FROM farmers_market.customer_purchases
LIMIT 10
```

market_date	customer_id	vendor_id	quantity	cost_to_customer_per_qty	quantity * cost_to_customer_per_qty
2019-03-02	4	8	2.00	4.00	8.0000
2019-03-02	10	8	1.00	4.00	4.0000
2019-03-09	12	8	1.00	4.00	4.0000
2019-03-09	5	9	1.00	16.00	16.0000
2019-03-09	1	9	1.00	18.00	18.0000
2019-03-02	2	4	4.60	2.00	9.2000
2019-03-02	3	4	8.40	2.00	16.8000
2019-03-02	4	4	1.40	2.00	2.8000
2019-03-09	4	4	9.90	2.00	19.8000
2019-03-02	1	1	1.00	5.50	5.5000

Simple Inline Calculations

To give the calculated column a meaningful name, we can create an alias by adding the keyword AS after the calculation and then specifying the new name.

```
SELECT
     market_date,
     customer_id,
     vendor_id,
     quantity * cost_to_customer_per_qty AS price
FROM farmers_market.customer_purchases
LIMIT 10
```

market_date	customer_id	vendor_id	price
2019-03-02	4	8	8.0000
2019-03-02	10	8	4.0000
2019-03-09	12	8	4.0000
2019-03-09	5	9	16.0000
2019-03-09	1	9	18.0000
2019-03-02	2	4	9.2000
2019-03-02	3	4	16.8000
2019-03-02	4	4	2.8000
2019-03-09	4	4	19.8000
2019-03-02	1	1	5.5000

Simple Inline Calculations

In MySQL syntax, the AS keyword is actually optional, so the following query will return the same results as the previous query.

```
SELECT
     market_date,
     customer_id,
     vendor_id,
     quantity * cost_to_customer_per_qty price
FROM farmers_market.customer_purchases
LIMIT 10
```

Inline Calculations: Rounding

A SQL function is a piece of code that takes inputs that you give it (which are called parameters), performs some operation on those inputs, and returns a value.

You can use functions inline in your query to modify the raw values from the database tables before displaying them in the output.

A SQL function call uses the following syntax:

```
FUNCTION_NAME([parameter 1],[parameter 2], . . . .[parameter n])
```

Inline Calculations: Rounding

To give an example of how functions are used in SELECT statements, we'll use the ROUND() function to round a number.

```
SELECT
     market_date,
     customer_id,
     vendor_id,
     ROUND(quantity * cost_to_customer_per_qty, 2) AS price
FROM farmers_market.customer_purchases
LIMIT 10
```

market_date	customer_id	vendor_id	price
2019-03-02	4	8	8.00
2019-03-02	10	8	4.00
2019-03-09	12	8	4.00
2019-03-09	5	9	16.00
2019-03-09	1	9	18.00
2019-03-02	2	4	9.20
2019-03-02	3	4	16.80
2019-03-02	4	4	2.80
2019-03-09	4	4	19.80
2019-03-02	1	1	5.50

Inline Calculations: Rounding

TIP The `ROUND()` function can also accept negative numbers for the second parameter, to round digits that are to the left of the decimal point. For example, `SELECT ROUND(1245, -2)` will return a value of 1200.

Inline Calculations: Concatenating Strings

There are also inline functions that can be used to modify string values in SQL, as well.

In our customer table, there are separate columns for each customer's first and last names

```
SELECT *
FROM farmers_m
LIMIT 5
```

customer_id	customer_first_name	customer_last_name	customer_zip
1	Jane	Connor	22801
2	Manuel	Diaz	22803
3	Bob	Wilson	22803
4	Deanna	Washington	22801
5	Abigail	Harris	22801

Inline Calculations: Concatenating Strings

Let's say we wanted to merge each customer's name into a single column that contains the first name, then a space, and then the last name.

We can accomplish that by using the CONCAT() function.

```
SELECT
     customer_id,
     CONCAT(customer_first_name, " ", customer_last_name) AS customer_
name
FROM farmers_market.customer
LIMIT 5
```

customer_id	customer_name
1	Jane Connor
2	Manuel Diaz
3	Bob Wilson
4	Deanna Washington
5	Abigail Harris

Inline Calculations: Concatenating Strings

Note that we can still add an ORDER BY clause and sort by last name first, even though the columns are merged together in the output:

```
SELECT
     customer_id,
     CONCAT(customer_first_name, " ", customer_last_name) AS customer_
name
FROM farmers_market.customer
ORDER BY customer_last_name, customer_first_name
LIMIT 5
```

customer_id	customer_name
7	Jessica Armenta
6	Betty Bullard
1	Jane Connor
2	Manuel Diaz
10	Russell Edwards

Inline Calculations: Concatenating Strings

It's also possible to nest functions inside other functions, which are executed by the SQL interpreter from the "inside" to the "outside."

```
SELECT
     customer_id,
     UPPER(CONCAT(customer_last_name, ", ", customer_first_name)) AS
customer_name
FROM farmers_market.customer
ORDER BY customer_last_name, customer_first_name
LIMIT 5
```

customer_id	customer_name
7	ARMENTA, JESSICA
6	BULLARD, BETTY
1	CONNOR, JANE
2	DIAZ, MANUEL
10	EDWARDS, RUSSELL

Inline Calculations: Concatenating Strings

NOTE Note that we did not sort on the new derived column alias `customer_name` here, but on columns that exist in the `customer` table. In some cases (depending on what database system you're using, which functions are used, and the execution order of your query) you can't reuse aliases in other parts of the query. It is possible to put some functions or calculations in the ORDER BY clause, to sort by the resulting value. Some other options for referencing derived values will be covered in later chapters.

Exercise

The following exercises refer to the customer table:

1. Write a query that returns everything in the customer table.

2. Write a query that displays all of the columns and 10 rows from the customer table, sorted by customer_last_name, then customer_first_name.

3. Write a query that lists all customer IDs and first names in the customer table, sorted by first_name.

The **WHERE** Clause

The WHERE clause is the part of the SELECT statement in which you list conditions that are used to determine which rows in the table should be included in the results set. In other words, the WHERE clause is used for filtering.

The WHERE clause goes after the FROM statement and before any GROUP BY, ORDER BY, or LIMIT statements in the SELECT query:

```
SELECT [columns to return]
FROM [table]
WHERE [conditional filter statements]
ORDER BY [columns to sort on]
```

The **WHERE** Clause

For example, to get a list of product IDs and product names that are in product category 1, you could use a conditional statement in the WHERE clause to select only rows from the product table in which the product_category_id is 1

```
SELECT
    product_id,
    product_name,
    product_category_id
FROM farmers_market.product
WHERE
    product_category_id = 1
LIMIT 5
```

product_id	product_name	product_category_id
1	Habanero Peppers - Organic	1
2	Jalapeno Peppers - Organic	1
3	Poblano Peppers - Organic	1
9	Sweet Potatoes	1
12	Baby Salad Lettuce Mix - Bag	1

The **WHERE** Clause

Let's say we wanted to print a report of everything a particular customer has ever purchased at the farmer's market, sorted by market date, vendor ID, and product ID.

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    product_id,
    quantity,
    quantity * cost_to_customer_per_qty AS price
FROM farmers_market.customer_purchases
WHERE customer_id = 4
ORDER BY market_date, vendor_id, product_id
LIMIT 5
```

market_date	customer_id	vendor_id	product_id	quantity	price
2019-03-02	4	4	9	1.40	2.8000
2019-03-02	4	8	4	2.00	8.0000
2019-03-09	4	1	10	1.00	5.5000
2019-03-09	4	4	9	9.90	19.8000
2019-03-09	4	7	12	2.00	6.0000

The **WHERE** Clause

What's happening behind the scenes is that each of the conditional statements (like "customer_id = 4") listed in the WHERE clause will evaluate to TRUE or FALSE for each row, and only the rows for which the combination of conditions evaluates to TRUE will be returned.

MARKET_ DATE	CUSTOMER_ ID	VENDOR_ID	PRODUCT_ ID	QUANTITY	PRICE	CONDITION: CUSTOMER_ ID = 4
2019-03-02	3	4	4	8.4	16.80	FALSE
2019-03-02	1	1	11	1.7	20.40	FALSE
2019-03-02	**4**	4	9	1.4	2.80	**TRUE**
2019-03-02	**4**	8	4	2.0	8.00	**TRUE**
2019-03-09	5	9	7	1.0	16.00	FALSE
2019-03-09	**4**	1	10	1.0	5.50	**TRUE**
2019-03-09	**4**	4	9	9.9	19.80	**TRUE**
2019-03-09	**4**	7	12	2.0	6.00	**TRUE**
2019-03-09	**4**	7	13	0.3	1.72	**TRUE**
2019-03-16	3	4	9	5.5	11.00	FALSE
2019-03-16	3	9	8	1.0	18.00	FALSE

The **WHERE** Clause

You can combine multiple conditions with boolean operators, such as “AND,” “OR,” or “AND NOT” between them in order to filter using multiple criteria in the WHERE clause.

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    product_id,
    quantity,
    quantity * cost_to_customer_per_qty AS price
FROM farmers_market.customer_purchases
WHERE customer_id = 3
    OR customer_id = 4
ORDER BY market_date, customer_id, vendor_id, product_id
```

market_date	customer_id	vendor_id	product_id	quantity	price
2019-03-02	3	4	9	8.40	16.8000
2019-03-02	4	4	9	1.40	2.8000
2019-03-02	4	8	4	2.00	8.0000
2019-03-09	4	1	10	1.00	5.5000
2019-03-09	4	4	9	9.90	19.8000
2019-03-09	4	7	12	2.00	6.0000
2019-03-09	4	7	13	0.30	1.7250
2019-03-16	3	4	9	5.50	11.0000
2019-03-16	3	9	8	1.00	18.0000

The **WHERE** Clause

What would happen if the WHERE clause condition were "customer_id = 3 AND customer_id = 4"?

MARKET_ DATE	CUSTOMER _ID	VENDOR _ID	PRICE	CONDITION: CUSTOMER_ ID = 3	AND	CONDITION: CUSTOMER_ ID = 4	ROW RETURNED?
2019-03-02	**3**	4	16.80	**TRUE**	*AND*	FALSE	*FALSE*
2019-03-02	1	1	20.40	FALSE	*AND*	FALSE	*FALSE*
2019-03-02	**4**	4	2.80	FALSE	*AND*	**TRUE**	*FALSE*
2019-03-02	**4**	8	8.00	FALSE	*AND*	**TRUE**	*FALSE*
2019-03-09	5	9	16.00	FALSE	*AND*	FALSE	*FALSE*

The **WHERE** Clause

Some people make the mistake of reading the logical AND operator the way we might request in English, “Give me all of the rows with customer IDs 3 and 4,” when what we really mean by that phrase is “Give me all of the rows where the customer ID is either 3 or 4,” which would require an OR operator in SQL.

When the AND operator is used, all of the conditions with AND between them must evaluate to TRUE for a row in order for that row to be returned in the query results.

The **WHERE** Clause

One example where you could use AND in a WHERE clause referring to only a single column is when you want to return rows with a range of values.

If someone requests “Give me all of the rows with a customer ID greater than 3 and less than or equal to 5,” the conditions would be written as “WHERE customer_id > 3 AND customer_id <= 5”

MARKET_ DATE	CUSTOMER _ID	VENDOR _ID	PRICE	CONDITION: CUSTOMER_ ID > 3	AND	CONDITION: CUSTOMER_ ID <= 5	ROW RETURNED?
2019-03-02	3	4	16.80	FALSE	*AND*	**TRUE**	*FALSE*
2019-03-02	1	1	20.40	FALSE	*AND*	**TRUE**	*FALSE*
2019-03-02	**4**	4	2.80	**TRUE**	*AND*	**TRUE**	***TRUE***
2019-03-02	**4**	8	8.00	**TRUE**	*AND*	**TRUE**	***TRUE***
2019-03-09	**5**	9	16.00	**TRUE**	*AND*	**TRUE**	***TRUE***
2019-03-09	**4**	1	5.50	**TRUE**	*AND*	**TRUE**	***TRUE***

The **WHERE** Clause

Let's try it in SQL, and see the output:

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    product_id,
    quantity,
    quantity * cost_to_customer_per_qty AS price
FROM farmers_market.customer_purchases
WHERE customer_id > 3
    AND customer_id <= 5
ORDER BY market_date, customer_id, vendor_id, product_id
```

market_date	customer_id	vendor_id	product_id	quantity	price
2019-03-02	4	4	9	1.40	2.8000
2019-03-02	4	8	4	2.00	8.0000
2019-03-09	4	1	10	1.00	5.5000
2019-03-09	4	4	9	9.90	19.8000
2019-03-09	4	7	12	2.00	6.0000
2019-03-09	4	7	13	0.30	1.7250
2019-03-09	5	9	7	1.00	16.0000

The **WHERE** Clause

Returning to the product table, let's examine a couple of queries and compare their output.

```
SELECT
    product_id,
    product_name
FROM farmers_market.product
WHERE
    product_id = 10
    OR (product_id > 3
    AND product_id < 8)
```

product_id	product_name
4	Banana Peppers - Jar
5	Whole Wheat Bread
6	Cut Zinnias Bouquet
7	Apple Pie
10	Eggs

The **WHERE** Clause

Returning to the product table, let's examine a couple of queries and compare their output.

```
SELECT
    product_id,
    product_name
FROM farmers_market.product
WHERE
    (product_id = 10
    OR product_id > 3)
    AND product_id < 8
```

product_id	product_name
4	Banana Peppers - Jar
5	Whole Wheat Bread
6	Cut Zinnias Bouquet
7	Apple Pie

The **WHERE** Clause: Multi-Column

WHERE clauses can also impose conditions using values in multiple columns.

For example, if we wanted to know the details of purchases made by customer 4 at vendor 7, we could use the following query:

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    quantity * cost_to_customer_per_qty AS price
FROM farmers_market.customer_purchases
WHERE
    customer_id = 4
    AND vendor_id = 7
```

market_date	customer_id	vendor_id	price
2019-03-09	4	7	6.0000
2019-03-09	4	7	1.7250

The **WHERE** Clause: Multi-Column

Let's try a WHERE clause that uses an OR condition to apply comparisons across multiple fields.

```
SELECT
    customer_id,
    customer_first_name,
    customer_last_name
FROM farmers_market.customer
WHERE
    customer_first_name = 'Carlos'
    OR customer_last_name = 'Diaz'
```

customer_id	customer_first_name	customer_last_name
2	Manuel	Diaz
17	Carlos	Diaz

The **WHERE** Clause: Multi-Column

And of course the conditions don’t both have to be “exact match” filters using equals signs.

```
SELECT *
FROM farmers_market.vendor_booth_assignments
WHERE
    vendor_id = 9
    AND market_date <= '2019-03-09'
ORDER BY market_date
```

vendor_id	booth_number	market_date
9	8	2019-03-02
9	8	2019-03-09

The **WHERE** Clause: More Ways to Filter

BETWEEN

We can use the BETWEEN keyword to see if a value, such as a date, is within a specified range of values.

```
SELECT *
FROM farmers_market.vendor_booth_assignments
WHERE
    vendor_id = 7
    AND market_date BETWEEN '2019-03-02' and '2019-03-16'
ORDER BY market_date
```

vendor_id	booth_number	market_date
7	11	2019-03-02
7	11	2019-03-09
7	11	2019-03-13

The **WHERE** Clause: More Ways to Filter

IN

We use the IN keyword and provide a comma-separated list of values to compare against.

customer_id	customer_first_name	customer_last_name
17	Carlos	Diaz
2	Manuel	Diaz
10	Russell	Edwards
3	Bob	Wilson

```
SELECT
    customer_id,
    customer_first_name,
    customer_last_name
FROM farmers_market.customer
WHERE
    customer_last_name IN ('Diaz' , 'Edwards', 'Wilson')
ORDER BY customer_last_name, customer_first_name
```

The **WHERE** Clause: More Ways to Filter

IN

This is instead of:

```
SELECT
    customer_id,
    customer_first_name,
    customer_last_name
FROM farmers_market.customer
WHERE
    customer_last_name = 'Diaz'
    OR customer_last_name = 'Edwards'
    OR customer_last_name = 'Wilson'
ORDER BY customer_last_name, customer_first_name
```

The **WHERE** Clause: More Ways to Filter

IN

Another use of the IN list comparison is if you’re searching for a person in the customer table, but don’t know the spelling of their name.

```
SELECT
    customer_id,
    customer_first_name,
    customer_last_name
FROM farmers_market.customer
WHERE
    customer_first_name IN ('Renee', 'Rene', 'Renée', 'René', 'Renne')
```

The **WHERE** Clause: More Ways to Filter

LIKE

Let's say that there was a farmer's market customer you knew as "Jerry," but you weren't sure if he was listed in the database as "Jerry" or "Jeremy" or "Jeremiah." All you knew for sure was that the first three letters were "Jer."

```
SELECT
    customer_id,
    customer_first_name,
    customer_last_name
FROM farmers_market.customer
WHERE
    customer_first_name LIKE 'Jer%'
```

customer_id	customer_first_name	customer_last_name
13	Jeremy	Gruber
18	Jeri	Mitchell

The **WHERE** Clause: More Ways to Filter

IS NULL

It’s often useful to find rows in the database where a field is blank or NULL.

```
SELECT *
FROM farmers_market.product
WHERE product_size IS NULL
```

product_id	product_name	product_size	product_category_id	product_qty_type
14	Red Potatoes	NULL	1	NULL

The **WHERE** Clause: More Ways to Filter

IS NULL

Keep in mind that “blank” and NULL are not the same thing in database terms.

```
SELECT *
FROM farmers_market.product
WHERE
    product_size IS NULL
    OR TRIM(product_size) = ''
```

TRIM() function removes excess spaces from the beginning or end of a string value

product_id	product_name	product_size	product_category_id	product_qty_type
14	Red Potatoes	NULL	1	NULL
15	Red Potatoes - Small		1	NULL

The **WHERE** Clause: More Ways to Filter

IS NULL

A Warning About Null Comparisons

You might wonder why the comparison operator `IS NULL` is used instead of equals NULL in the previous section. NULL is not actually a value, it's the absence of a value, so it can't be compared to any existing value. If your query were filtered to `WHERE product_size = NULL`, no rows would be returned, even though there is a record with a NULL product_size, because nothing "equals" NULL, even NULL.

The **WHERE** Clause: More Ways to Filter

IS NULL

This is important for other types of comparisons as well.

```
SELECT
    market_date,
    transaction_time,
    customer_id,
    vendor_id,
    quantity
FROM farmers_market.customer_purchases
WHERE
    customer_id = 1
    AND vendor_id = 7
    AND quantity > 1
```

market_date	transaction_time	customer_id	vendor_id	quantity
2019-03-09	08:42:00	1	7	2.20
2019-03-20	08:25:00	1	7	3.10

The **WHERE** Clause: More Ways to Filter

IS NULL

This is important for other types of comparisons as well.

```
SELECT
    market_date,
    transaction_time,
    customer_id,
    vendor_id,
    quantity
FROM farmers_market.customer_purchases
WHERE
    customer_id = 1
    AND vendor_id = 7
    AND quantity <= 1
```

market_date	transaction_time	customer_id	vendor_id	quantity

NULL values aren’t comparable to numbers in that way, there is a record that is never returned when there’s a numeric comparison used, because it has a NULL value in the quantity field.

The **WHERE** Clause: More Ways to Filter

IS NULL

You can see that if you run this query:

```
SELECT
    market_date,
    transaction_time,
    customer_id,
    vendor_id,
    quantity
FROM farmers_market.customer_purchases
WHERE
    customer_id = 1
    AND vendor_id = 7
```

market_date	transaction_time	customer_id	vendor_id	quantity
2019-03-09	08:42:00	1	7	2.20
2019-03-09	08:43:00	1	7	NULL
2019-03-20	08:25:00	1	7	3.10

Ideally, the database should be designed so that the quantity value for a purchase record isn't allowed to be NULL.

You could use the condition "[field name] IS NOT NULL" in the WHERE clause.

The **WHERE** Clause: Filtering Using Subqueries

When the IN list comparison was demonstrated earlier, it used a hard-coded list of values. What if you wanted to filter to a list of values that was returned by another query?

There is a way to do that in SQL, using a subquery (a query inside a query).

```
SELECT market_date, market_rain_flag
FROM farmers_market.market_date_info
WHERE market_rain_flag = 1
```

market_date	market_rain_flag
2019-03-20	1
2019-03-23	1
2019-03-30	1

Now we need to use the list of dates generated by that query to return purchases made on those dates.

The **WHERE** Clause: Filtering Using Subqueries

Note that when using a query in an IN comparison, you can only return the field you're comparing to, so we will not include the market_rain_flag field in the following subquery.

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    quantity * cost_to_customer_per_qty price
FROM farmers_market.customer_purchases
WHERE
    market_date IN
        (
        SELECT market_date
        FROM farmers_market.market_date_info
        WHERE market_rain_flag = 1
        )
LIMIT 5
```

market_date	customer_id	vendor_id	price
2019-03-20	7	7	9.0000
2019-03-23	4	7	9.0000
2019-03-30	12	7	3.0000
2019-03-20	1	7	17.8250
2019-03-23	4	7	13.8000

Creating results that depend on data in more than one table can also be accomplished a JOIN.

Exercise

1. Write a query that returns all customer purchases of product IDs 4 and 9.

2. Write two queries, one using two conditions with an AND operator, and one using the BETWEEN operator, that will return all customer purchases made from vendors with vendor IDs between 8 and 10 (inclusive).

3. Can you think of two different ways to change the final query in the slides so it would return purchases from days when it wasn't raining?

رواد مصر الرقمية

وزارة الاتصالات
وتكنولوجيا المعلومات

Q&A

Questions and answers

وزارة الاتصـــــالات
وتكنولوجيا المعلومات

رواد مصر الرقمية

Thanks!