UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

3250 Foundations of Data Science

Module 10: Databases and SQL

Course Plan

Module Titles
Module 1 – Introduction to Data Science
Module 2 – Introduction to Python
Module 3 – NumPy
Module 4 – Pandas
Module 5 – Data Collection and Cleaning
Module 6 – Descriptive Statistics and Visualization
Module 7 – Workshop
Module 8 – Time Series
Module 9 – Introduction to Regression and Classification
Current Focus: Module 10 – Databases and SQL
Current Focus: Module 11 – Data Privacy and Security
Module 12 – Term Project Presentations (no content)

Learning Outcomes for this Module

- Discuss relational database management systems
- Describe what schemas are
- Learn SQL commands to manipulate and retrieve data

Topics for this Module

- **10.1** Introduction to SQL and Relational Databases
- **10.2** SQL Basics
- **10.3** SQL Joins
- **10.4** Python with SQLite
- **10.5** Installing SQLite
- **10.6** Resources and Wrap-up

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 1

Introduction to SQL and Relational Databases

A Short History of Relational Database Technology

- Relational databases are organized based on the relational model of data (proposed in 1970 by E. F. Codd)
- The first RDBMS was called System R and was created by IBM
- SQL (Structured Query Language) was created to manage the data stored in System R
- In 1979, a company called Relational Software released Oracle V2 to commercialize SQL

Relational vs. Non-Relational Databases

Relational Databases

- Data organized in tables and columns
- Have a concept of primary and foreign keys
- Mature technology (ORACLE, SQL Server, MySQL, etc.)
- Work well for data with predefined structure
- Great for analytics (i.e. select customers who regularly spend in a certain category)

Non-Relational Databases

- “NoSQL” databases like MongoDB, don’t require designing data relations (no need for a data model)
- Used as a “data store” for data than can change daily
- Used in big data and real-time web applications
- Many varieties such as key-value, document, graph

Relational vs. Non-Relational Databases (Examples)

Relational Databases

- ORACLE
- MS SQL Server
- Sybase
- MySQL
- MS Access
- IBM DB2
- Teradata
- Netezza
- SAP Hana
- PostgresSQL
- MariaDB (Google)

Non-Relational databases

- MongoDB
- Cassandra
- HBase
- ORACLE NoSQL database
- Redis
- InfiniteGraph
- Alchemy Database
- ArangoDB
- OrientDB

What is SQL?

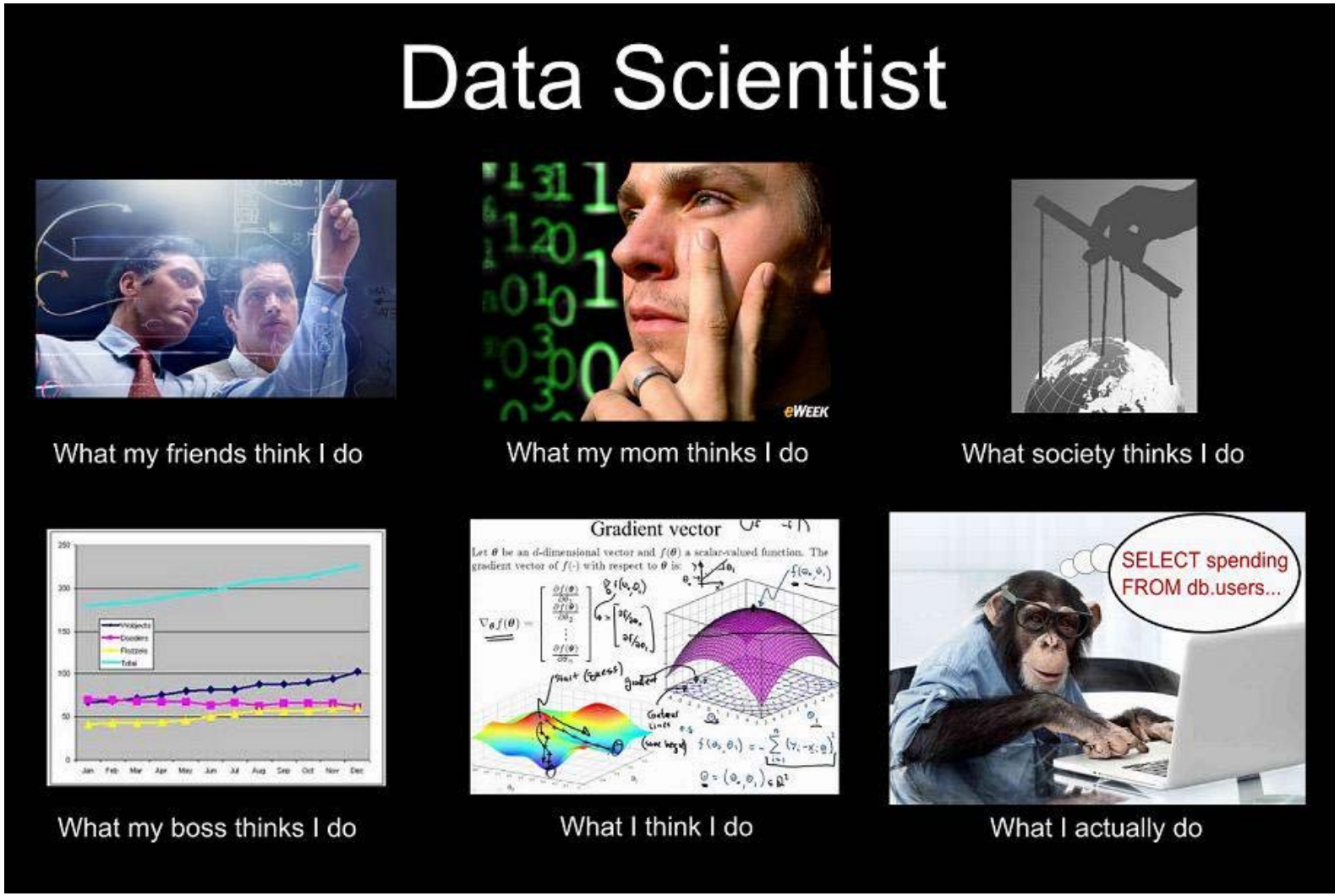
SQL (Structured Query Language) is a standard language for accessing and manipulating databases

- SQL is a database language, used to update databases, execute queries, and manage permissions
- It was designed for relational database management systems such as: MS SQL Server, MySQL, MS Access, Oracle, Informix, Sybase, DB2, and other databases
- SQL is an ANSI (American National Standards Institute) standard
- SQL was initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s.

Why should we learn SQL?

- Is it important for a Data Scientist to be self sufficient
  - **You need to grab your own data!**
- A major part of a Data Scientist's work day is involved around data retrieval and preparation for analysis
- Data is often stored in databases that use SQL, or a version of SQL

Why should we learn SQL? (cont'd)



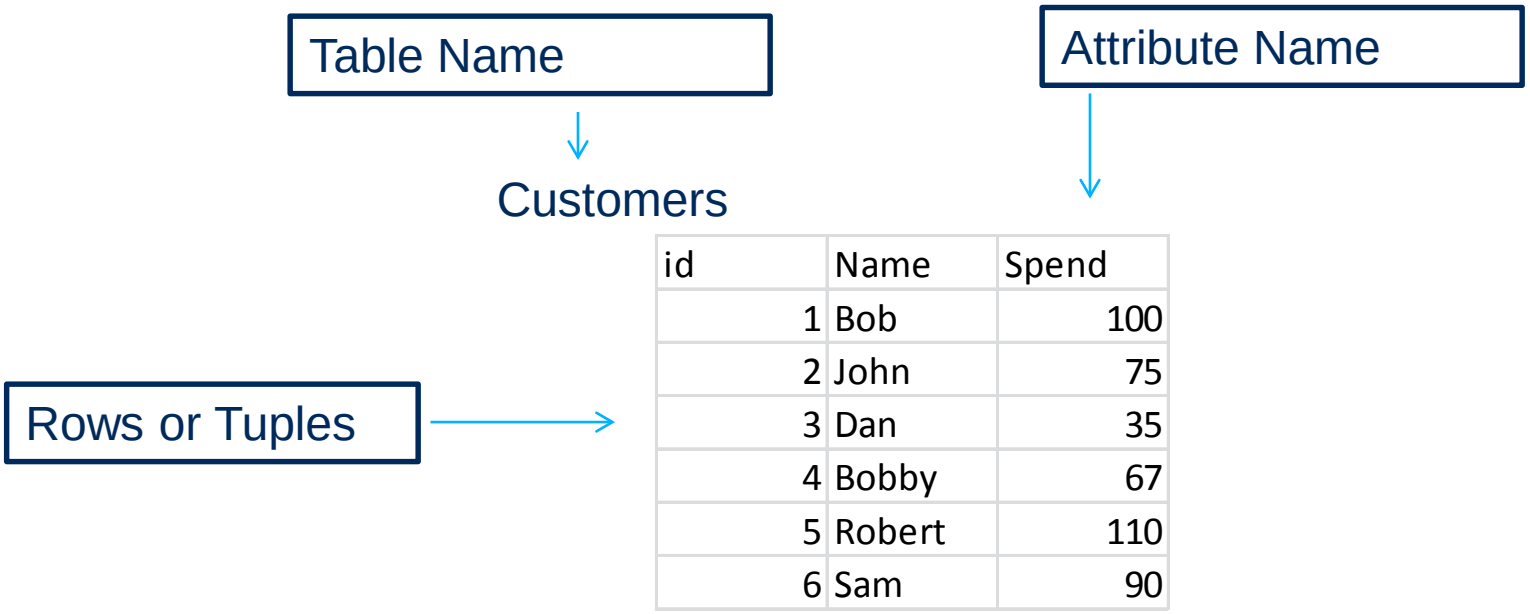
Who works with SQL?

- Database developers and administrators
- Business Analysts
- Programmers
- Web developers
- Data Scientists
- Statisticians
- Analysts who need to retrieve data from a database

What can SQL do?

- Execute queries against databases
- Retrieve data which can then be used for further analysis
  - For example, a data scientists could retrieve sales data from a databases, and further analyse it using R, Python, or other tools
- Join tables and views to create new tables and views
- Create new data/records/tables/views
- Drop data/records/tables/views
- Insert, update, create, delete data/records/tables/views
- Create/delete/update views
- Set permissions on tables, procedures, views, etc.
- Create stored procedures (this is used for automation)
  - For example, we want to score customers on a monthly basis into segments (based on a model that we have developed)

Table Basics



id	Name	Spend
1	Bob	100
2	John	75
3	Dan	35
4	Bobby	67
5	Robert	110
6	Sam	90

A relational database system contains one or more objects: tables and views. The data is stored in tables, made of columns (attributes) and rows (tuples).

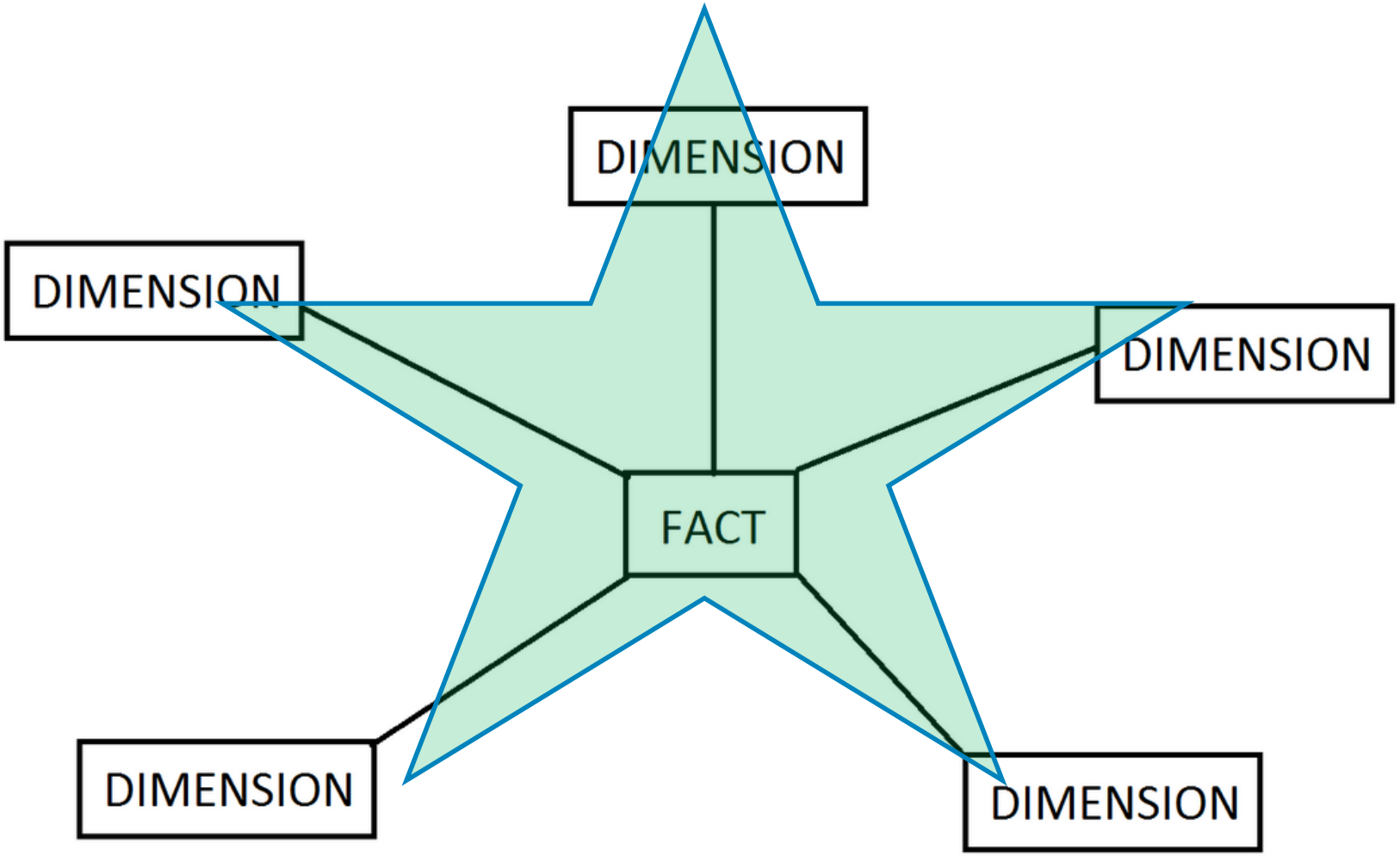
Terminology

Row	Tuple or Record
Column	Field or Attribute
Table	Data set (columns & rows)
View	Derived (from tables & or views)
Result Set	Derived as response to query

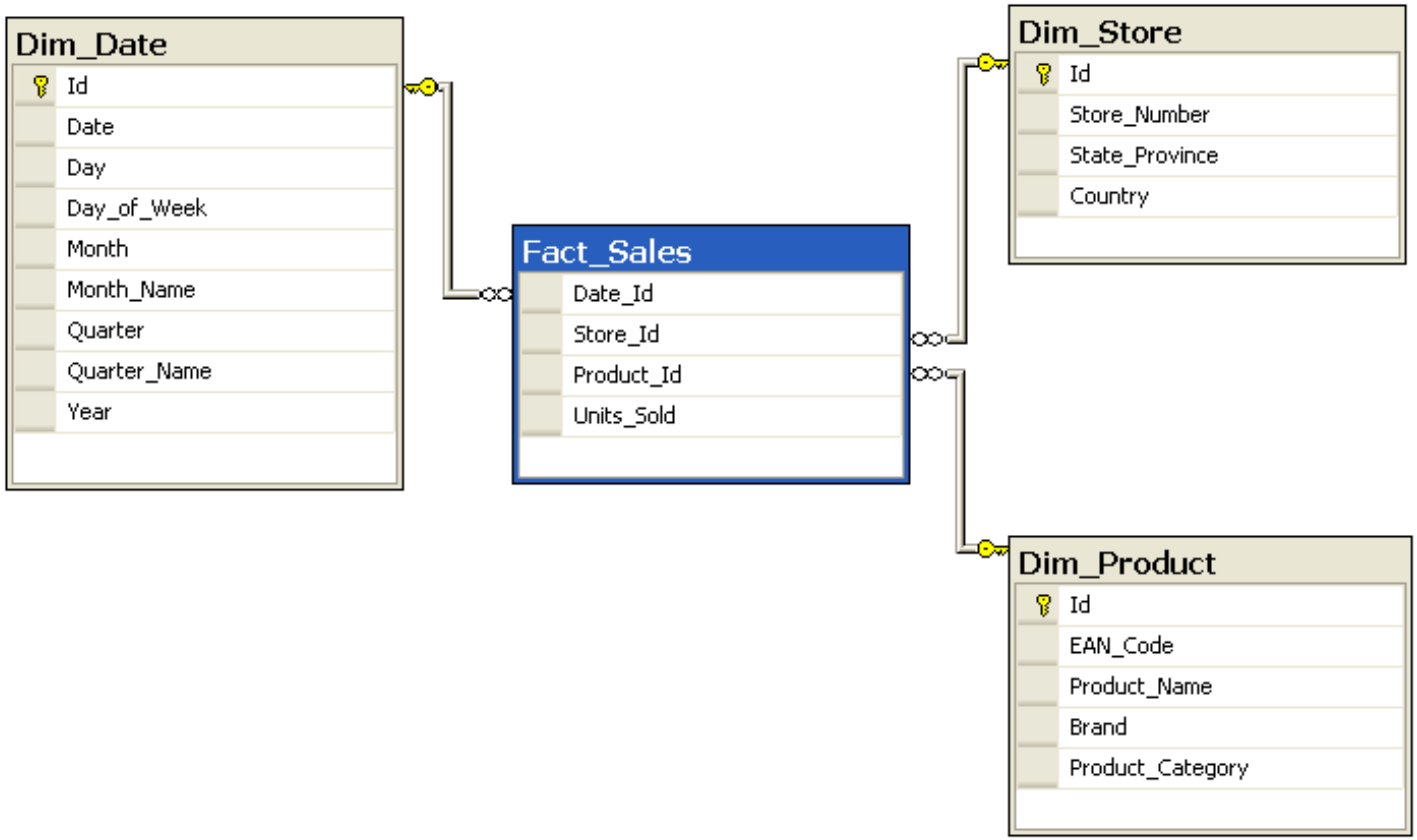
What is a Schema?

- A relational database schema is the definition of the tables, columns and relationships that make up a relational database
- A schema can be often depicted visually in modeling software
- A relational database schema helps analysts to organize and understand the structure of a database

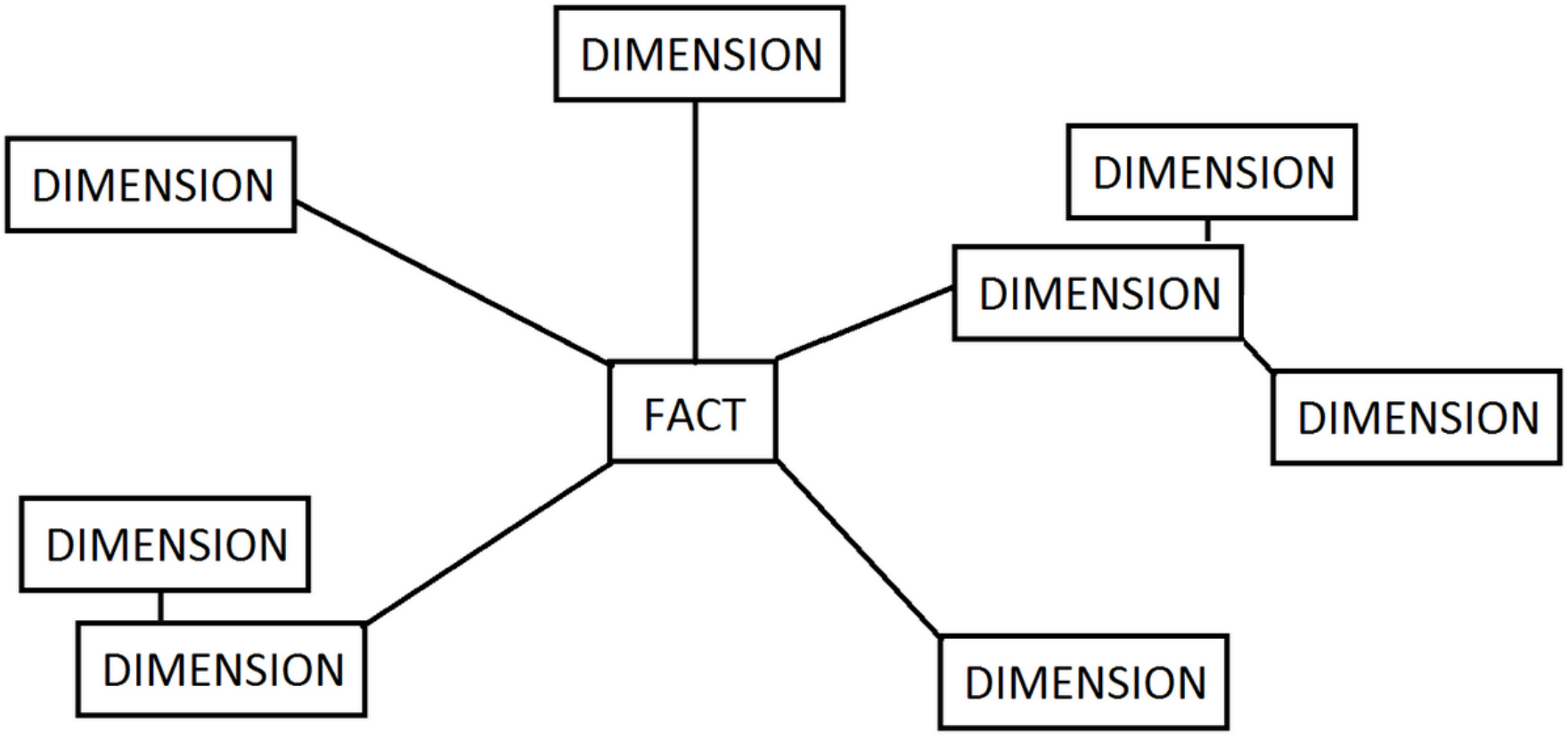
Star Schema



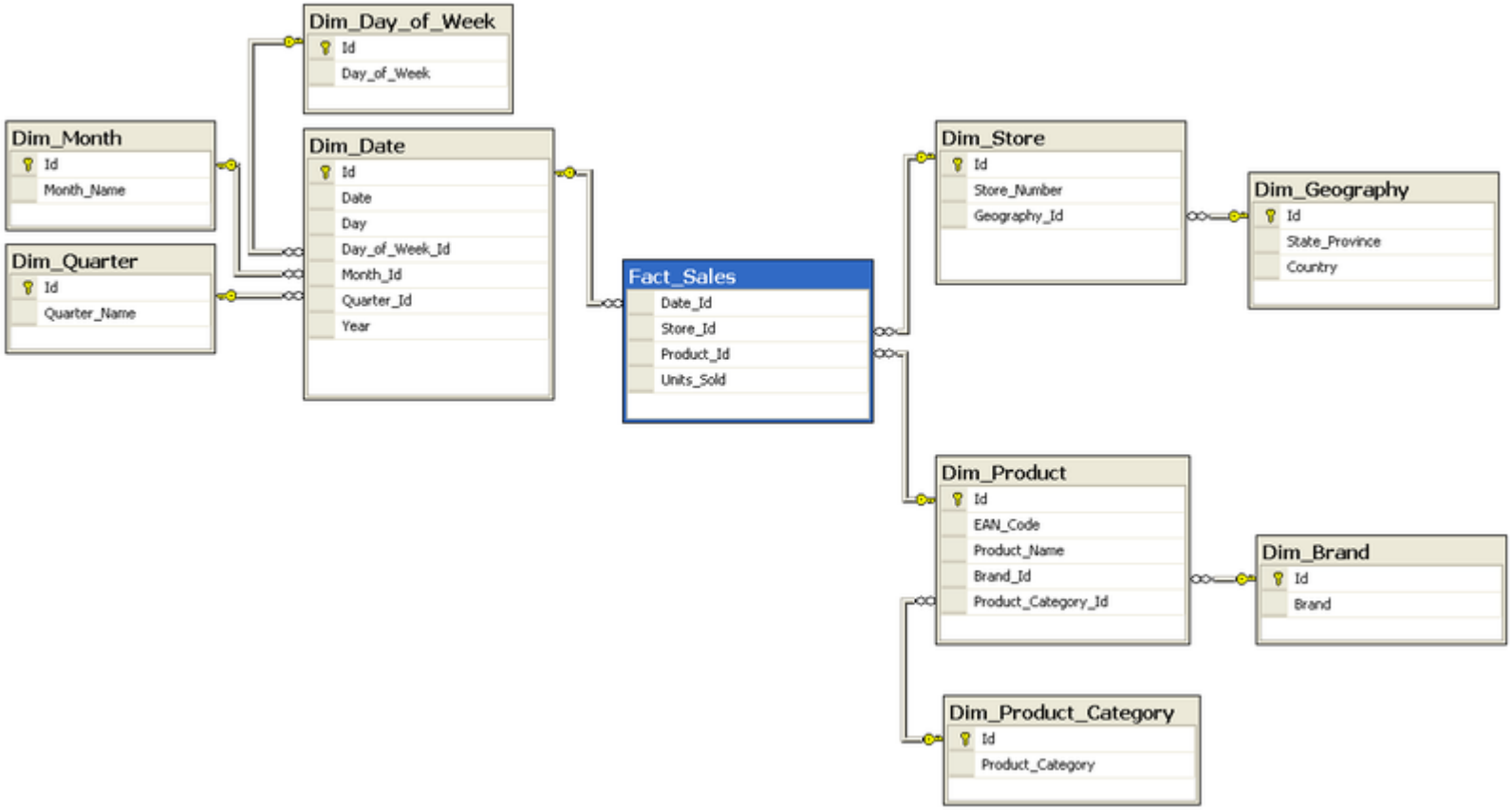
Star Schema (cont'd)



Snowflake Schema



Snowflake Schema (cont'd)



Primary Key

- **A primary key is a column (or combination of columns) designated to uniquely identify all table records**.
  - must contain a unique value for each row
    - customer id
    - order number
    - transaction id
  - cannot contain null values.

- A primary key is either an existing table column or a column that is specifically generated by the database according to a defined sequence

Foreign Key

- **A foreign key is a column (field, or collection of columns (fields) in one table that uniquely identifies a row of another table**

- The foreign key is defined in a second table, but it refers to the primary key in the first table.

Example

`ArtistId` is the primary key in the **artists** table

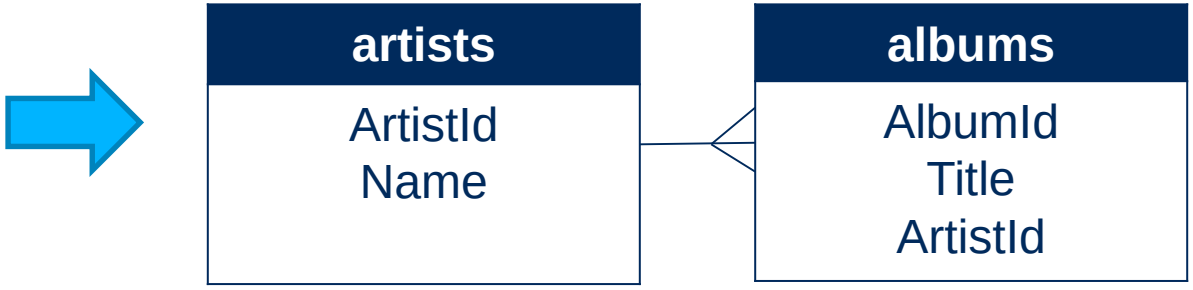
`ArtistId` is a foreign key in the **albums** table

`AlbumId` is the primary key in the **albums** table

Database Table Relationships

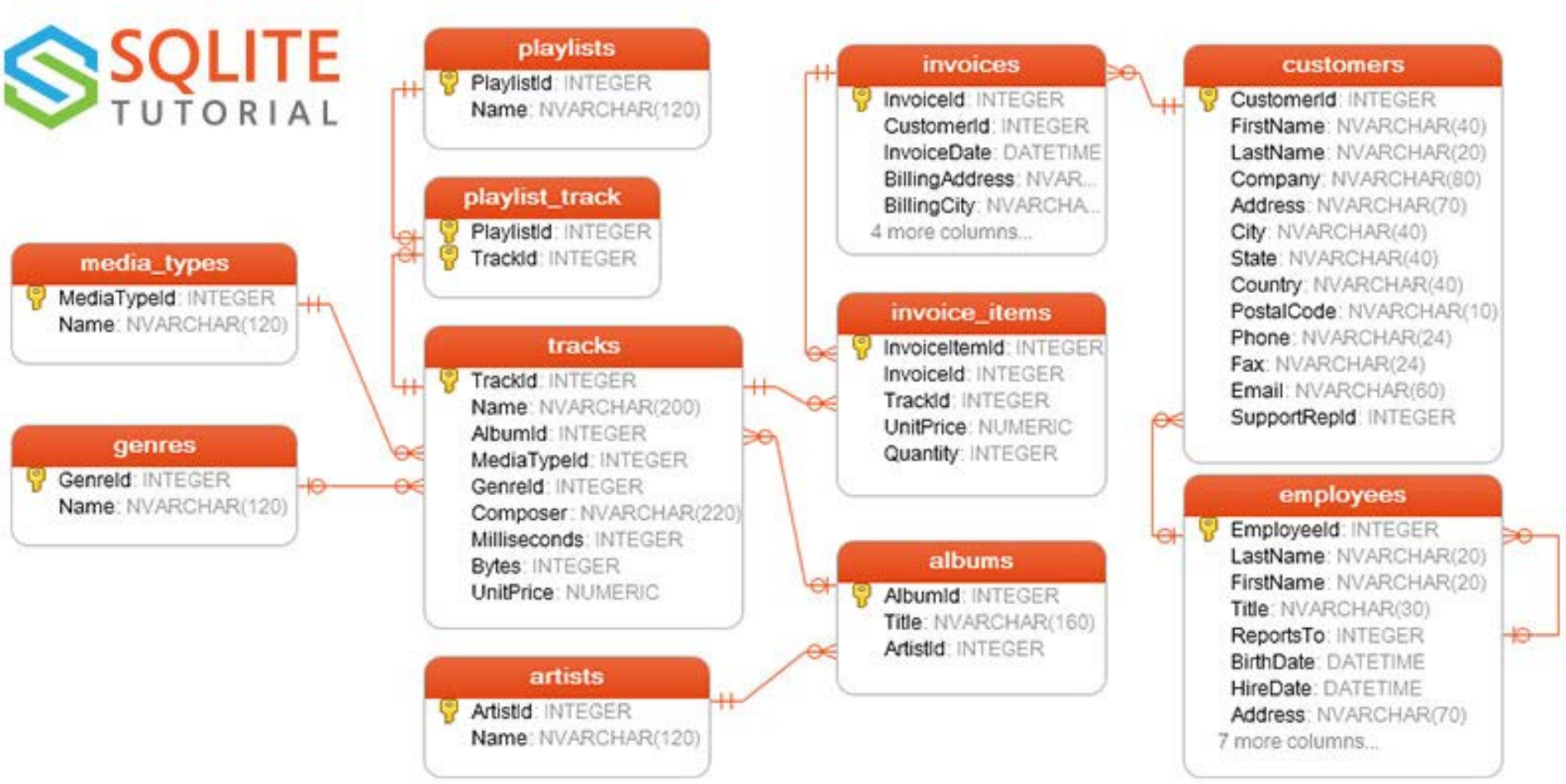
- One-to-one

- One-to-Many



- Many-to-Many

Example


SQLITE
TUTORIAL
playlists
PlaylistId: INTEGER
Name: NVARCHAR(120)
playlist_track
PlaylistId: INTEGER
TrackId: INTEGER
media_types
MediaTypeId: INTEGER
Name: NVARCHAR(120)
genres
GenreId: INTEGER
Name: NVARCHAR(120)
tracks
TrackId: INTEGER
Name: NVARCHAR(200)
AlbumId: INTEGER
MediaTypeId: INTEGER
GenreId: INTEGER
Composer: NVARCHAR(220)
Milliseconds: INTEGER
Bytes: INTEGER
UnitPrice: NUMERIC
artists
ArtistId: INTEGER
Name: NVARCHAR(120)
invoices
InvoiceId: INTEGER
CustomerId: INTEGER
InvoiceDate: DATETIME
BillingAddress: NVAR...
BillingCity: NVARCHA...
4 more columns...
invoice_items
InvoiceItemId: INTEGER
InvoiceId: INTEGER
TrackId: INTEGER
UnitPrice: NUMERIC
Quantity: INTEGER
albums
AlbumId: INTEGER
Title: NVARCHAR(160)
ArtistId: INTEGER
customers
CustomerId: INTEGER
FirstName: NVARCHAR(40)
LastName: NVARCHAR(20)
Company: NVARCHAR(80)
Address: NVARCHAR(70)
City: NVARCHAR(40)
State: NVARCHAR(40)
Country: NVARCHAR(40)
PostalCode: NVARCHAR(10)
Phone: NVARCHAR(24)
Fax: NVARCHAR(24)
Email: NVARCHAR(60)
SupportRepId: INTEGER
employees
EmployeeId: INTEGER
LastName: NVARCHAR(20)
FirstName: NVARCHAR(20)
Title: NVARCHAR(30)
ReportsTo: INTEGER
BirthDate: DATETIME
HireDate: DATETIME
Address: NVARCHAR(70)
7 more columns...

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 2

SQL Basics

SQL Data Manipulation

- Data manipulation is essential for SQL tables
  - allows you to modify an already created table with new information, update or delete already existing values
  - For example, with the INSERT statement, you can add new rows to an already existing table.
  - New rows can contain information from the start, or can be with a NULL value (you can create an empty table with a certain “structure” and then populate the table with values)

Common SQL statements

- **SELECT** - extracts data from a database
- **UPDATE** - updates data in a database
- **DELETE** - deletes data from a database
- **INSERT INTO** - inserts new data into a database
- **CREATE DATABASE** - creates a new database
- **ALTER DATABASE** - modifies a database
- **CREATE TABLE** - creates a new table
- **ALTER TABLE** - modifies a table
- **DROP TABLE** - deletes a table
- **CREATE INDEX** - creates an index (search key)
- **DROP INDEX** - deletes an index

SQL Queries

SQL queries are the most common and essential SQL operations, and are executed with the “SELECT” statement.

Here is the general structure:

```
CREATE TABLE table_name AS
        SELECT ….. [grab the columns needed]
                FROM ….[tables/views]
        WHERE …..[apply relevant conditions/filters]
        ORDER BY …..[you can sort the output]
```

CREATE TABLE

Customers

id	Name	Spend
1	Bob	100
2	John	75
3	Dan	35
4	Bobby	67
5	Robert	110
6	Sam	90

Resulting Output

id	Name	Spend
2	John	75
4	Bobby	67
6	Sam	90

```
CREATE TABLE Customers_Two AS
    SELECT * FROM Customers
    WHERE Spend BETWEEN 40 AND 95
    ORDER BY ID;
```

CREATE TABLE (cont'd)

This creates a table with Null values.

- name of the table is : phonebook
- 4 columns created, each is a character field.

```
CREATE TABLE  phonebook
        (phone VARCHAR(32),
        firstname VARCHAR(32),
        lastname VARCHAR(32),
        address VARCHAR(64));
```

CREATE TABLE (cont'd)

```
CREATE TABLE prodsales
 (product CHAR(3),
  mnth    SMALLINT,
  sales   MONEY) ;
```

In this table, sales amounts (sales) are stored by month (mnth) and product code (product). The mnth column stores an integer value ranging from 1 (for January) to 12 (for December).

DELETE

The DELETE statement is used to delete rows from a table

```
DELETE FROM table_name
WHERE some_column=some_value;
```

```
DELETE FROM Customers
        WHERE Spend BETWEEN 40 AND 75;
```

DROP

The DROP TABLE statement is used to delete a table

DROP TABLE table_name;

DROP TABLE Customers;

DROP DATABASE database_name;
deletes a database

TRUNCATE

What if we only want to delete the data inside the table, and not the table itself?

We can use the TRUNCATE TABLE statement:

TRUNCATE TABLE table_name;

UPDATE

The UPDATE statement is used to update records in a table

```
UPDATE table_name
SET column1=value1,column2=value2,...
WHERE some_column=some_value;
```

SQL Operators

Operator	Description	Example
=	Equal to	Name = 'FRANK'
<>	Not equal to (many DBMSs accept != in addition to <>)	Dept <> 'Sales'
>	Greater than	Hire_Date > '2015-01-31'
<	Less than	Bonus < 40000.00
>=	Greater than or equal	Dependents >= 3
<=	Less than or equal	Rate <= 0.05
BETWEEN	Between an inclusive range	Cost BETWEEN 100.00 AND 500.00
IN	Equal to one of multiple possible values	DeptCode IN (100, 201, 309)
IS *or* IS NOT	Compare to null (missing data)	Address IS NOT NULL
IS NOT DISTINCT FROM	Is equal to value or both are nulls (missing data)	Debt IS NOT DISTINCT FROM - Receivables
AS	Used to change a field name when viewing results	SELECT employee AS 'department1'
LIKE	Match a character pattern	First_Name LIKE 'Will%'

SQL SELECT Statement

	InvoiceId	CustomerId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingPostalCode	Total
1	1	2	2009-01-01 00:00:00	Theodor-Heuss-Straß...	Stuttgart	*NULL*	Germany	70174	1.98
2	2	4	2009-01-02 00:00:00	Ullevålsveien 14	Oslo	*NULL*	Norway	0171	3.96
3	3	8	2009-01-03 00:00:00	Grétrystraat 63	Brussels	*NULL*	Belgium	1000	5.94
4	4	14	2009-01-06 00:00:00	8210 111 ST NW	Edmonton	AB	Canada	T6G 2C7	8.91
5	5	23	2009-01-11 00:00:00	69 Salem Street	Boston	MA	USA	2113	13.86
6	6	37	2009-01-19 00:00:00	Berger Straße 10	Frankfurt	*NULL*	Germany	60316	0.99
7	7	38	2009-02-01 00:00:00	Barbarossastraße 19	Berlin	*NULL*	Germany	10779	1.98
8	8	40	2009-02-01 00:00:00	8, Rue Hanovre	Paris	*NULL*	France	75002	1.98
9	9	42	2009-02-02 00:00:00	9, Place Louis Barthou	Bordeaux	*NULL*	France	33000	3.96

SQL SELECT Statement (cont'd)

invoices

	CustomerId	Total
1	2	1.98
2	4	3.96
3	8	5.94
4	14	8.91
5	23	13.86
6	37	0.99
7	38	1.98
8	40	1.98
9	42	3.96

SELECT CustomerId, Total FROM invoices;

This would only retrieve 2 columns: CustomerId and Total from the table 'invoices', but all rows

SQL SELECT Statement (cont'd)

artists

	ArtistId	Name
8	8	Audioslave
9	9	BackBeat
10	10	Billy Cobham
11	11	Black Label Society
12	12	Black Sabbath
13	13	Body Count
14	14	Bruce Dickinson
15	15	Buddy Guy
16	16	Caetano Veloso

Resulting Output

	ArtistId	Name
1	11	Black Label Society
2	12	Black Sabbath
3	169	Black Eyed Peas

SELECT * **FROM** artists **WHERE** Name **LIKE** 'Bl%';

This would grab all columns where Name starts with 'Bl'

Strings must be in single quotes

SQL SELECT Statement (cont'd)

artists

	ArtistId	Name
1	1	AC/DC
2	2	Accept
3	3	Aerosmith
4	4	Alanis Morissette
5	5	Alice In Chains
6	6	Antônio Carlos Jobim
7	7	Apocalyptica
8	8	Audioslave
9	9	BackBeat

Resulting Output

	ArtistId	Name
1	1	AC/DC
2	228	Leonard Bernstein & New York Philharmonic
3	259	The 12 Cellists of The Berlin Philharmonic

SELECT * FROM artists **WHERE** Name **LIKE** '%c';

This would grab rows where Name ends with 'c'

Strings must be in single quotes.

LIKE

LIKE operator performs a basic pattern-matching using wildcard characters.

For Microsoft SQL Server, the wildcard characters are defined as follows:

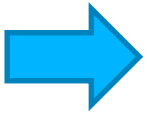
- _ (underscore) matches any single character
- % matches a string of one or more characters
- [ ] matches any single character within the specified range (e.g. [a-f]) or set (e.g. [abcdef]).
- [^] matches any single character not within the specified range (e.g. [^a-f]) or set (e.g. [^abcdef]).

Example

To find all artists whose name does not start with ‘B' or ‘D', we would write

```
SELECT *  FROM artists
        WHERE (Name NOT LIKE ‘B%')
                AND (Name NOT LIKE ‘D%');
```

ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith
4	Alanis Morissette
5	Alice In Chains
6	Antônio Carlos Jobim
7	Apocalyptica
8	Audioslave
9	BackBeat
10	Billy Cobham
11	Black Label Society
12	Black Sabbath
13	Body Count
14	Bruce Dickinson
15	Buddy Guy
16	Caetano Veloso
17	Chico Buarque



ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith
4	Alanis Morissette
5	Alice In Chains
6	Antônio Carlos Jobim
7	Apocalyptica
8	Audioslave
16	Caetano Veloso
17	Chico Buarque
18	Chico Science & Nação Zumbi
19	Cidade Negra
20	Cláudio Zoli

More LIKE Examples

Assume you are working with the ‘artists’ table that contains Name Column:

- **WHERE** Name **LIKE** '_2' finds all two-letter artist names that end with ‘2' (e.g. U2).
- **WHERE** Name **LIKE** '%nic' finds all artists and groups whose name ends with ‘nic’ (e.g. ‘Leonard Bernstein & New York Philharmonic’)
- **WHERE** Name **LIKE** '%nic%' finds all artists and groups where name includes ’nic' anywhere in the name (for example, in addition to the names ending with ‘nic’, it will find ‘Mônica Marianno’).
- **WHERE** Name **LIKE** '[JT]im' finds three-letter names that end with 'im' and begin with either 'J' or 'T'
- **WHERE** Name **LIKE** ’M[^a]%' finds all names beginning with ’M' where the following (second) letter is not ’a'.

IN Statement

artists

ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith
4	Alanis Morissette
5	Alice In Chains
6	Antônio Carlos Jobim
7	Apocalyptica
8	Audioslave
9	BackBeat
10	Billy Cobham
11	Black Label Society
12	Black Sabbath

Resulting Output

ArtistId	Name
3	Aerosmith
4	Alanis Morissette
5	Alice In Chains

SELECT * **FROM** artists **WHERE** ArtistId **IN** (3,4,5);

BETWEEN Statement

invoices

CustomerId	Total
2	1.98
4	3.96
8	5.94
14	8.91
23	13.86
37	0.99
38	1.98
40	1.98
42	3.96
46	5.94

Resulting Output

CustomerId	Total
37	0.99
16	0.99
54	0.99
33	0.99
12	0.99
50	0.99
29	0.99
8	0.99
46	0.99
25	0.99

```
SELECT CustomerId, Total FROM invoices
    WHERE Total BETWEEN 0 AND 1;
```

CASE Statement

SQL has the **case/when/then/else/end** expression (works like IF-ELSE-THEN in other languages), that comes in very handy in recoding/creating new variables

Example:

```
CASE WHEN n > 0  THEN 'positive'
        WHEN n < 0 THEN 'negative'
        ELSE 'zero'
END
```

Example

tracks

TrackId	Name	AlbumId	MediaTypeId	GenreId	Composer	Milliseconds
1	For Those About To Rock (W...	1	1	1	Angus Young, Malcolm Young, Bria...	343719
2	Balls to the Wall	2	2	1	*NULL*	342562
3	Fast As a Shark	3	2	1	F. Baltes, S. Kaufman, U. Dirkscnei...	230619
4	Restless and Wild	3	2	1	F. Baltes, R.A. Smith-Diesel, S. Kau...	252051
5	Princess of the Dawn	3	2	1	Deaffy & R.A. Smith-Diesel	375418
6	Put The Finger On You	1	1	1	Angus Young, Malcolm Young, Bria	205662

```
SELECT  TrackId, Name
CASE WHEN Milliseconds < 60000 THEN 'short'
WHEN Milliseconds > 60000 AND Milliseconds < 300000
THEN 'medium'
ELSE 'long'
END category
FROM tracks;
```

TrackId	Name	category
1	For Those About To Rock (We ...	long
2	Balls to the Wall	long
3	Fast As a Shark	medium
4	Restless and Wild	medium
5	Princess of the Dawn	long
6	Put The Finger On You	medium
7	Let's Get It Up	medium

DISTINCT

If a table contains multiple records, one way to remove duplicates is to use DISTINCT in the SELECT statement:

```
SELECT DISTINCT city
        FROM customers;
```

will return 53 rows

City
São José dos Campos
Stuttgart
Montréal
Oslo
Prague
Vienne
Brussels
Copenhagen
São Paulo
Rio de Janeiro
Brasília
Edmonton
Vancouver
Mountain View

SQL Aggregate Functions

SQL Aggregate Functions

SQL aggregate functions return a single value, calculated from values in a column:

- **AVG()** - Returns the average value
- **COUNT()** - Returns the number of rows
- **FIRST()** - Returns the first value
- **LAST()** - Returns the last value
- **MAX()** - Returns the largest value
- **MIN()** - Returns the smallest value
- **SUM()** - Returns the sum

Aggregate Function Example

invoices

InvoiceId	CustomerId	Total
1	2	1.98
2	4	3.96
3	8	5.94
4	14	8.91
5	23	13.86
6	37	0.99
7	38	1.98
8	40	1.98
9	42	3.96
10	46	5.94
11	52	8.91

Resulting Output

Total_Spend	Ave_Spend
2328.600000000004	5.651941747572825

```
SELECT Sum(Total) AS Total_Spend,
       Avg(Total) AS Ave_Spend
  FROM invoices;
```

Aggregate Functions & GROUP BY

```
SELECT
        column_name,
        aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name;
```

SQL Scalar Functions

SQL scalar functions return a single value, based on the input value

- **UCASE()** - Converts a field to upper case
- **LCASE()** - Converts a field to lower case
- **MID()** - Extracts characters from a text field
- **LEN()** - Returns the length of a text field
- **ROUND()** - Rounds a numeric field to the number of decimals specified
- **NOW()** - Returns the current system date and time
- **FORMAT()** - Formats how a field is to be displayed

UPPER() and LOWER()

UPPER and **UCASE** are equivalent string manipulation functions that manipulate CHARACTER string data and convert lowercase characters in a string to uppercase. (LOWER() and LCASE() are used to convert the string to lower case.)

```
SELECT
        TOP 10 UPPER (firstname) as fname, lastname
FROM [dbo].Students;
```

dbo is the name of your database

fname is a new variable created which would be the upper case of firstname

HAVING

The HAVING clause was added because the WHERE keyword could not be used with aggregate functions.

SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value;

Checking the Result

When you work with a new dataset, it often helps to do some "sense checking"

- How many records?
  - Select count(*) from [dbo].datasetname;
- What does the data look like?
  - Select top 100 * from [dbo].datasetname;
- Is there a unique primary key?
- What are the means, frequencies, min/max observations of our columns of interest?

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 2 – Section 3

SQL Joins

SQL Joins

Joins in SQL allow the user to select columns from one or more tables and create a set (result) that could be stored in another table, view or be used as required (for example exported out for further analysis).

ANSI-standard SQL specifies five types of join:

INNER, LEFT OUTER, RIGHT OUTER, FULL OUTER and CROSS.

* Note that a table can JOIN to itself.

AS (Alias)

When joining or selecting tables, it is important to specify aliases (this is done so that you don't have to write out explicitly the table and database names):

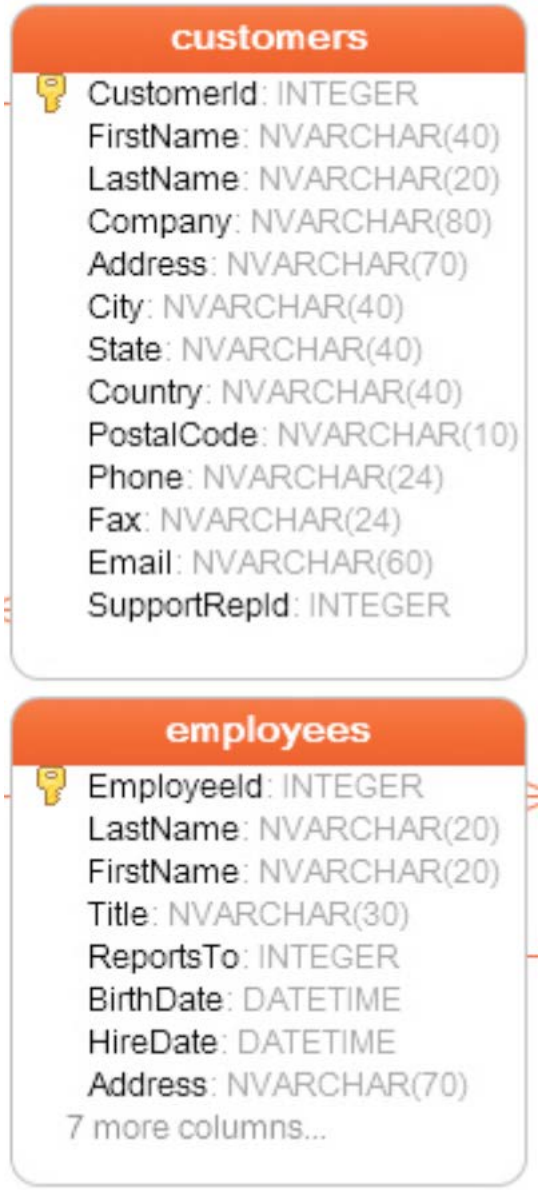
```
SELECT column_name AS column_alias
FROM table_name
```

or

```
SELECT column_name
FROM table_name  AS table_alias
```

Example of Column and Table Aliases

```
SELECT
FirstName AS Cust_FName,
LastName AS Cust_LName
FROM customers AS Cust
```



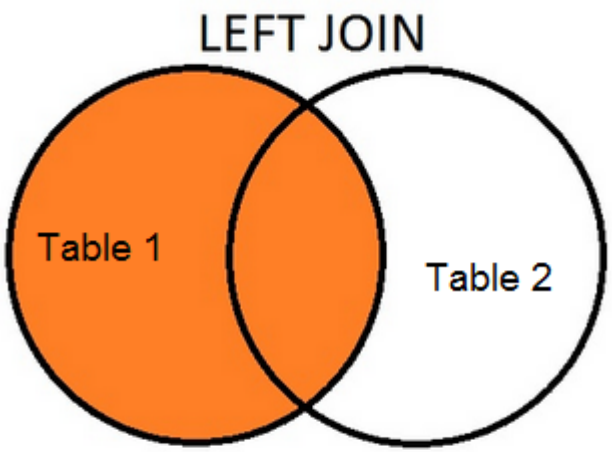
So….What is a JOIN?

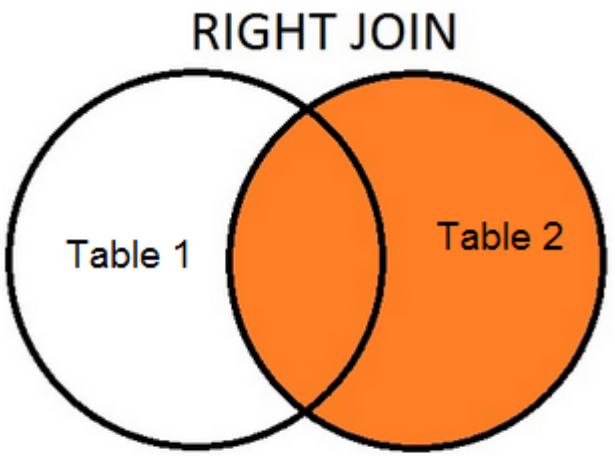
A SQL **JOIN** clause combines columns from one or more tables in a relational database.

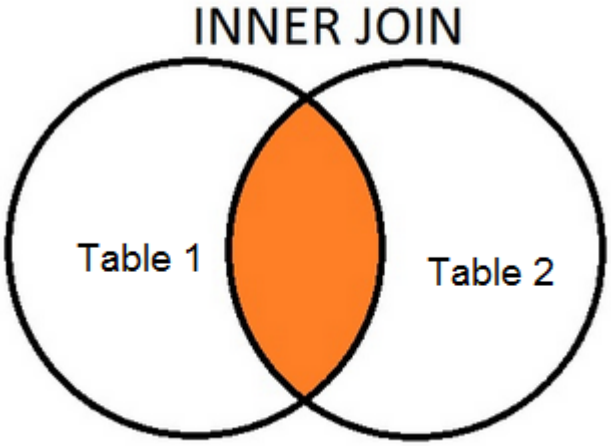
It creates a set that can be saved as a table or used as it is. A **JOIN** is a means for combining columns from one or more tables by using values common to each.

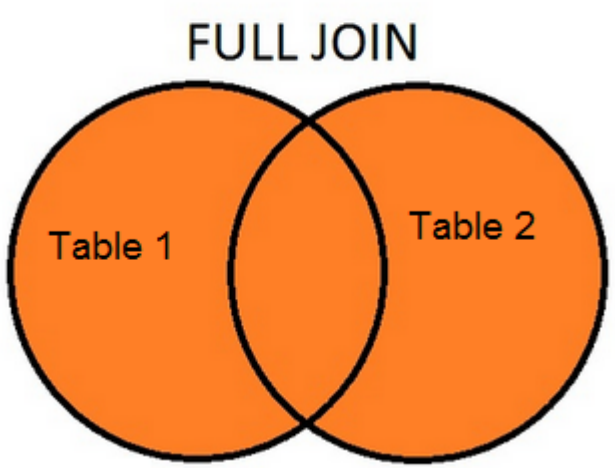
It allows us to create more descriptive views of the data.

Types of JOINs as Venn Diagrams









Types of JOINs

(INNER) JOIN: returns rows when there is a match in both tables.

LEFT (OUTER) JOIN: returns all rows from the left table, even if there are no matches in the right table. Fills missing data with NULLs.

RIGHT (OUTER) JOIN: returns all rows from the right table, even if there are no matches in the left table. Fills missing data with NULLs.

FULL (OUTER) JOIN: combines left and right outer joins. The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

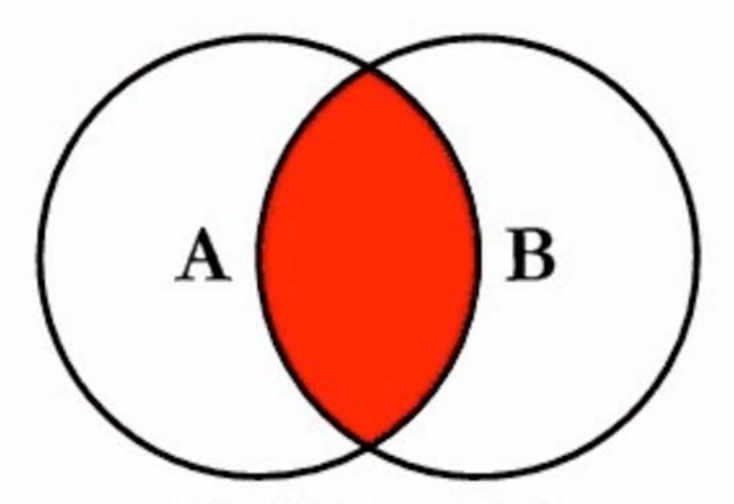
SELF JOIN: is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

CROSS JOIN: returns the Cartesian product of the sets of records from the two or more joined tables.

INNER JOIN is the same as JOIN

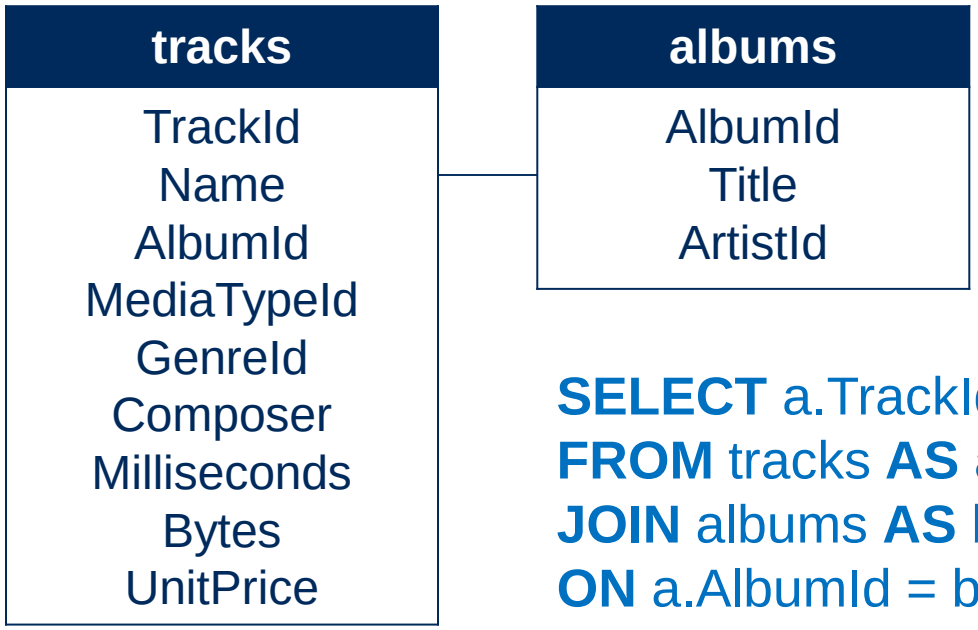
SELECT *column_name(s)*
FROM *tableA*
INNER JOIN *tableB*
ON *tableA.column_name=tableB.column_name*;

The same as:



SELECT *column_name(s)*
FROM *tableA*
JOIN *tableB*
ON *tableA.column_name=tableB.column_name*;

SQL JOIN Example



SQLite assumes that JOIN is an INNER JOIN.

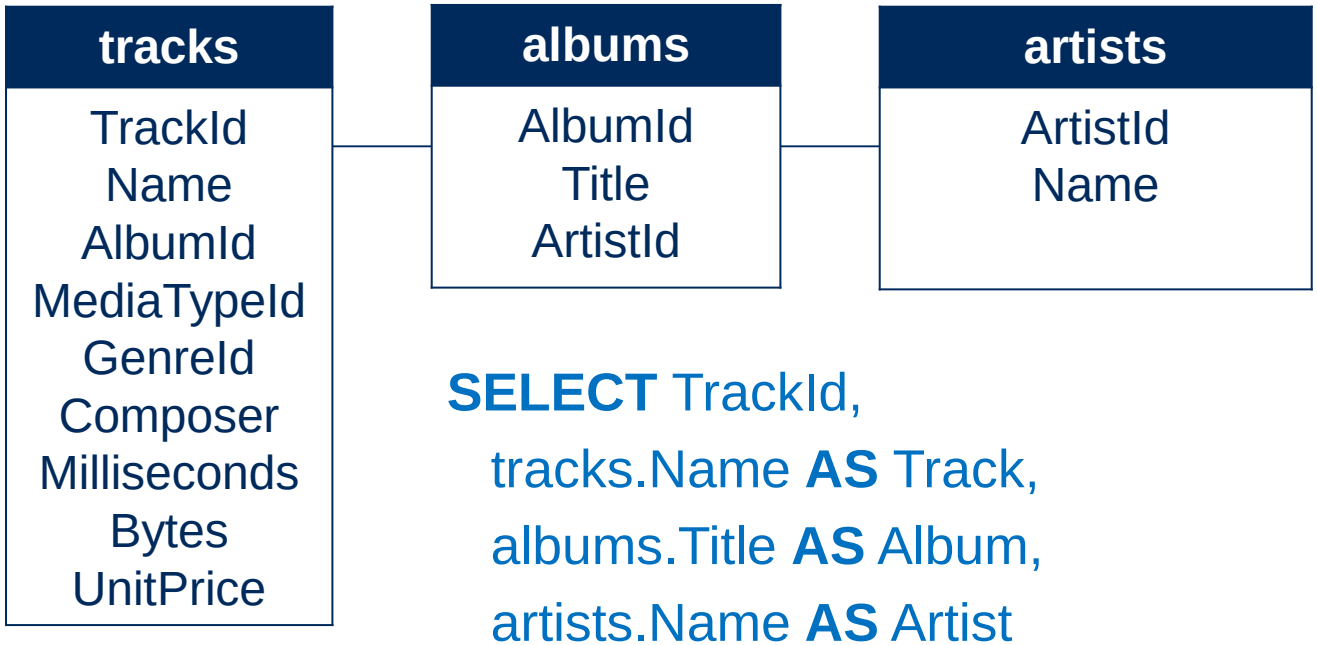
```
SELECT a.TrackId, a.Name, b.Title
FROM tracks AS a
JOIN albums AS b
ON a.AlbumId = b.AlbumId;
```

The following gives the same result:

```
SELECT a.TrackId, a.Name, b.Title
FROM tracks a
JOIN albums b
ON a.AlbumId = b.AlbumId;
```

NOTE: when you alias a table you can omit the 'AS'

INNER JOIN 3 Tables Example




```
SELECT TrackId,
    tracks.Name AS Track,
    albums.Title AS Album,
    artists.Name AS Artist
FROM tracks
INNER JOIN albums ON albums.AlbumId =
tracks.AlbumId
INNER JOIN artists ON artists.ArtistId =
albums.ArtistId
```

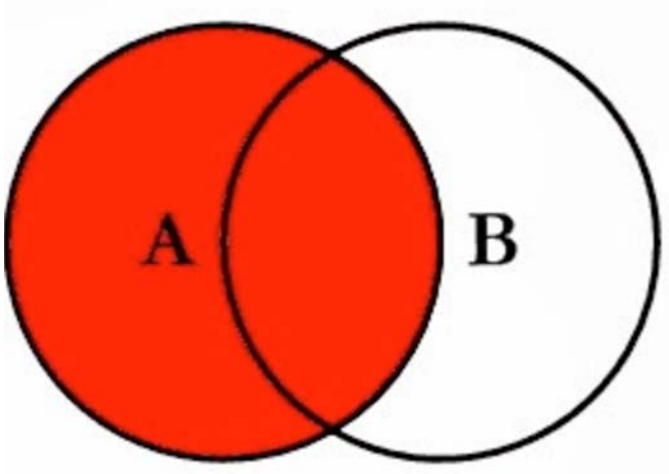
LEFT JOIN Example

artists
ArtistId
Name

albums
AlbumId
Title
ArtistId

The ORDER BY keyword is used to sort the result-set by one or more columns.

```
SELECT
   artists.ArtistId,
   albumId
FROM artists
LEFT JOIN albums
ON albums.ArtistId = artists.ArtistId
ORDER BY AlbumId;
```



SELF JOIN Example

employees
EmployeeId
LastName
FirstName
Title
ReportsTo
BirthDate
HireDate
Address
City
State
Country
PostalCode
Phone
Fax
Email

```
SELECT
m.FirstName || ' ' || m.LastName AS 'Manager',
e.FirstName || ' ' || e.LastName AS 'Direct report'
FROM employees e
INNER JOIN employees m
ON m.EmployeeId = e.ReportsTo
ORDER BY Manager;
```

NOTE: CEO does not report to anyone. If we use LEFT JOIN instead of INNER JOIN, we will see his name in the output with NULL in the Manager column

UNION and UNION ALL

SELECT *column_name(s)* **FROM** *table1*
UNION
SELECT *column_name(s)* **FROM** *table2*;

SELECT *column_name(s)* **FROM** *table1*
UNION ALL
SELECT *column_name(s)* **FROM** *table2*;

Note: The UNION operator selects only distinct values by default

UNION and UNION ALL Example

```
SELECT FirstName, LastName
FROM employees
UNION ALL
SELECT FirstName, LastName
FROM employees;
```

- The statement above returns 16 rows.
- The same statement, but with UNION instead of UNION ALL, returns 8 rows

Indexes

Indexes allow the database application to find data fast without reading the whole table.

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 2 – Section 4

Python with SQLite

Connect to the Database and Make Changes to the Table

- Create a connection object that represents a database:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

- Once the connection to the database is opened, create a **`Cursor`** object and call its **`execute()`** method to perform SQL commands:

```
c = conn.cursor()
c.execute("INSERT INTO MediaTypeId
VALUES ('6','New Type')")
```

- To save the changes to the table:

```
conn.commit()
```

- Close connection to the database:

```
conn.close()
```

Retrieve Data from the Database

- Retrieve values from the table using SELECT statement:

```
t = ('AC/DC')
c.execute('SELECT * FROM artists WHERE
Name=?', t)
print(c.fetchone())
```

- To retrieve a single matching row:

```
c.fetchone()
```

- To retrieve a list of the matching rows:

```
c.fetchall()
```

Resources

- sqlite3 — DB-API 2.0 interface for SQLite databases:
- The Python SQL Toolkit and Object Relational Mapper – SQLAlchemy:

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 2 – Section 5

Installing SQLite

Installing SQLite on your Laptop

- Go to sqlitestudio and choose the right installation for your laptop from the table “Latest stable release (3.1.1)”
- Unzip the downloaded file into a directory of your choice
- On Windows, run “SQLiteStudio.exe”
- On Mac, double-click “sqlitestudio-3.1.1.dmg” which you downloaded, and move “SQLiteStudio.app” into “Applications”

Installing SQLite Studio

Download MacOS X binary (ix86_64)
Version 3.1.1
(25.2 MB)

SQLite Studio

About | Gallery | Download | Changelog | Forum | Wiki | Bugs & Ideas | The Wall | Contact | Links

Lastest stable release (3.1.1):

Distribution	Platform	Size	Version	Link
Windows	32-bit	16.4MB	3.1.1	sqlitestudio-3.1.1.zip
Linux	64-bit	18.7MB	3.1.1	sqlitestudio-3.1.1.tar.xz
MacOSX	64-bit (ix86_64)	25.2MB	3.1.1	sqlitestudio-3.1.1.dmg
Sources (zip)	Independent	9.0MB	3.1.1	sqlitestudio-3.1.1.zip
Sources (tar.gz)	Independent	8.3MB	3.1.1	sqlitestudio-3.1.1.tar.gz

..::All files - click here::..

..::Old, unsupported versions (2.x.x) - click here::..

News RSS

Version 3.1.1 released!
2016-11-02
Among tons of bugfixes, there are some new features, such as plugins supporting wxSQLite and System.Data.SQLite databases (latter one only under Windows), batch importing with import() (and its import_*() family) function, support for "Row Value" from recent SQLite 3.15.0 and others - see ChangeLog for full list of changes.

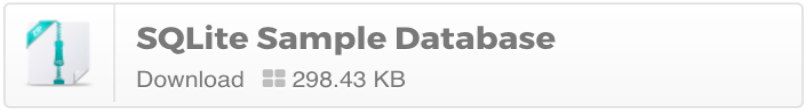
3.1.0 released!
2016-06-11
Next major release - it brings SQLCipher plugin and

Downloading Sample Database

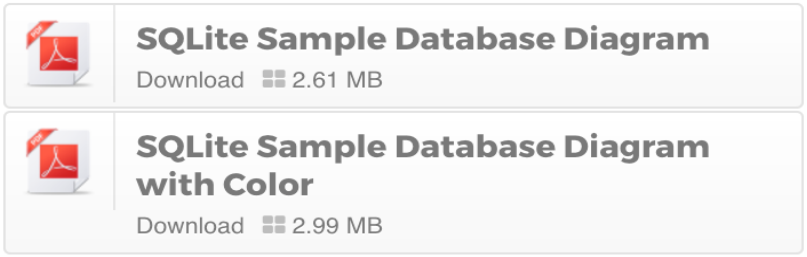
www.sqlitetutorial.net/sqlite-sample-database/

Download SQLite sample database

You can download the SQLite sample database using the following link.



In case you want to have the database diagram for reference, you can download both black&white and color versions in PDF format.



How to connect to SQLite sample database

The sample database file is ZIP format, therefore, you need to extract it to a folder, for example, C:\sqlite\db. The name of the file is chinook.db

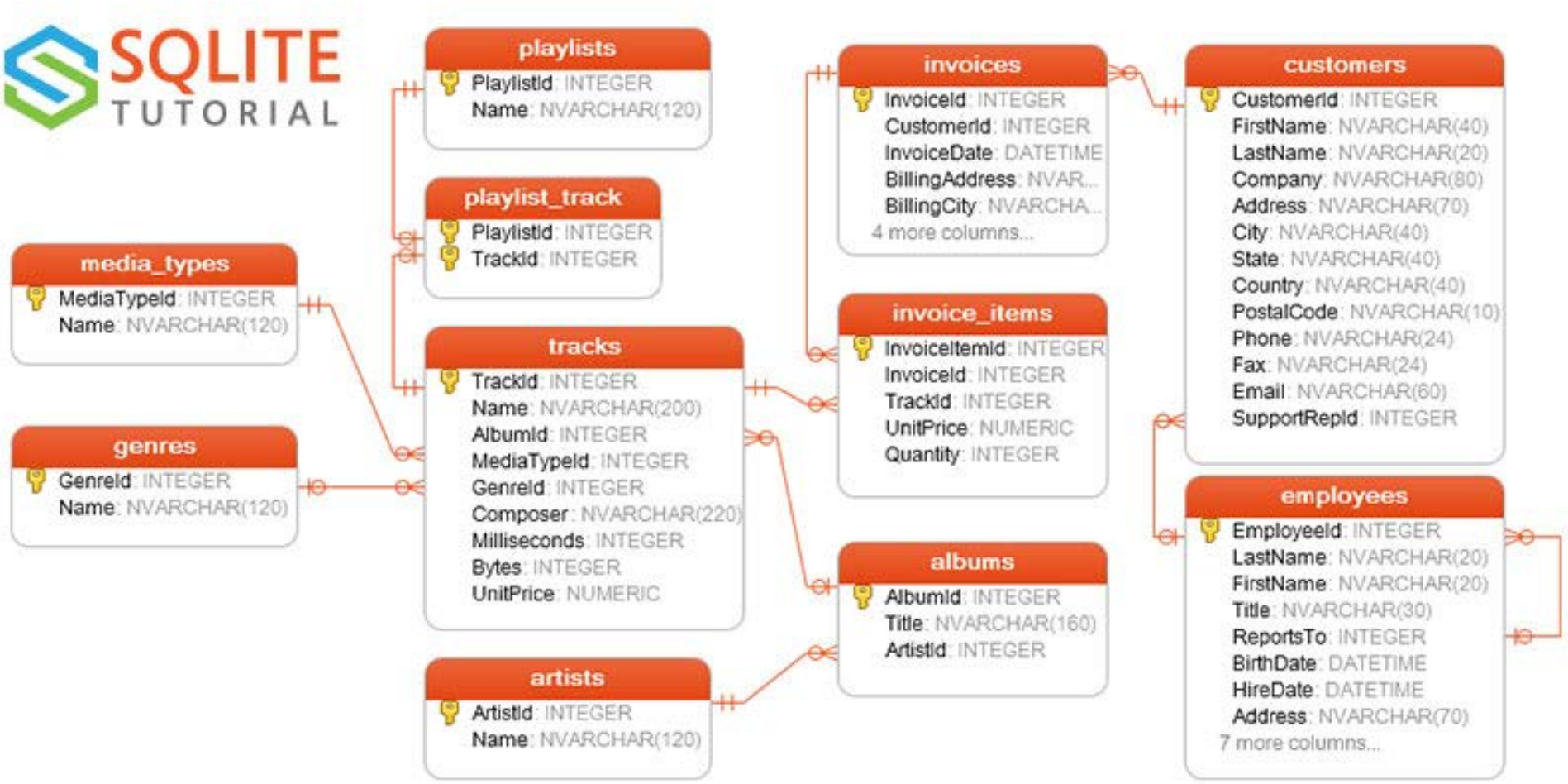
Loading Sample Database

- After you downloaded and unzipped the sample database, open SQLite Studio
- From Database menu, select “Add a Database”
- In the next window, “File” field, navigate to the folder where you saved the sample database, file “chinook.db” and click “OK”
- The database will be loaded into the SQLite database and opened in SQLite Studio
- The home page for the Chinook database project is here - github.com/lerocha/chinook-database

What is in the Database “Chinook”?

- Data Model
  - The Chinook data model represents a digital media store, including tables for artists, albums, media tracks, invoices and customers.
- Sample Data
  - Media related data was created using real data from an iTunes Library. Customer and employee information was created using fictitious names and addresses. Sales information is auto generated using random data for a four year period.
- The name Chinook:
  - The name of this sample database was based on the NorthWind database. Chinooks are winds where the Canadian Prairies and Great Plains meet various mountain ranges and are most prevalent over southern Alberta.

Database Schema



Source

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 2 – Section 6

Resources and Wrap-up

Resources

- W3Schools SQL tutorial
- DZone Beginner's Guide to the True Order of SQL Operations
- Books:
  - "Learning SQL" By Alan Beaulieu
  - "SQL in 10 Minutes" By Ben Forta
  - "SQL Queries for Mere Mortals" By John L. Viescas and Michael J. Hernandez
  - "SQL Cookbook" By Anthony Molinaro
  - "SQL Database Programming" (2015 Edition) By Chris Fehily

Next Class

- Data Security and Privacy

Follow us on social

Join the conversation with us online:

facebook.com/uoftscs

@uoftscs

linkedin.com/company/university-of-toronto-school-of-continuing-studies

@uoftscs

Any questions?

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Thank You

Thank you for choosing the University of Toronto School of Continuing Studies