

**CS 398: Application Development**

# Working with Databases

Relational databases; SQL; SQLite.

# Databases

Instead of storing data in files, we can choose to store everything in a **database**: a system designed for organizing and managing data.

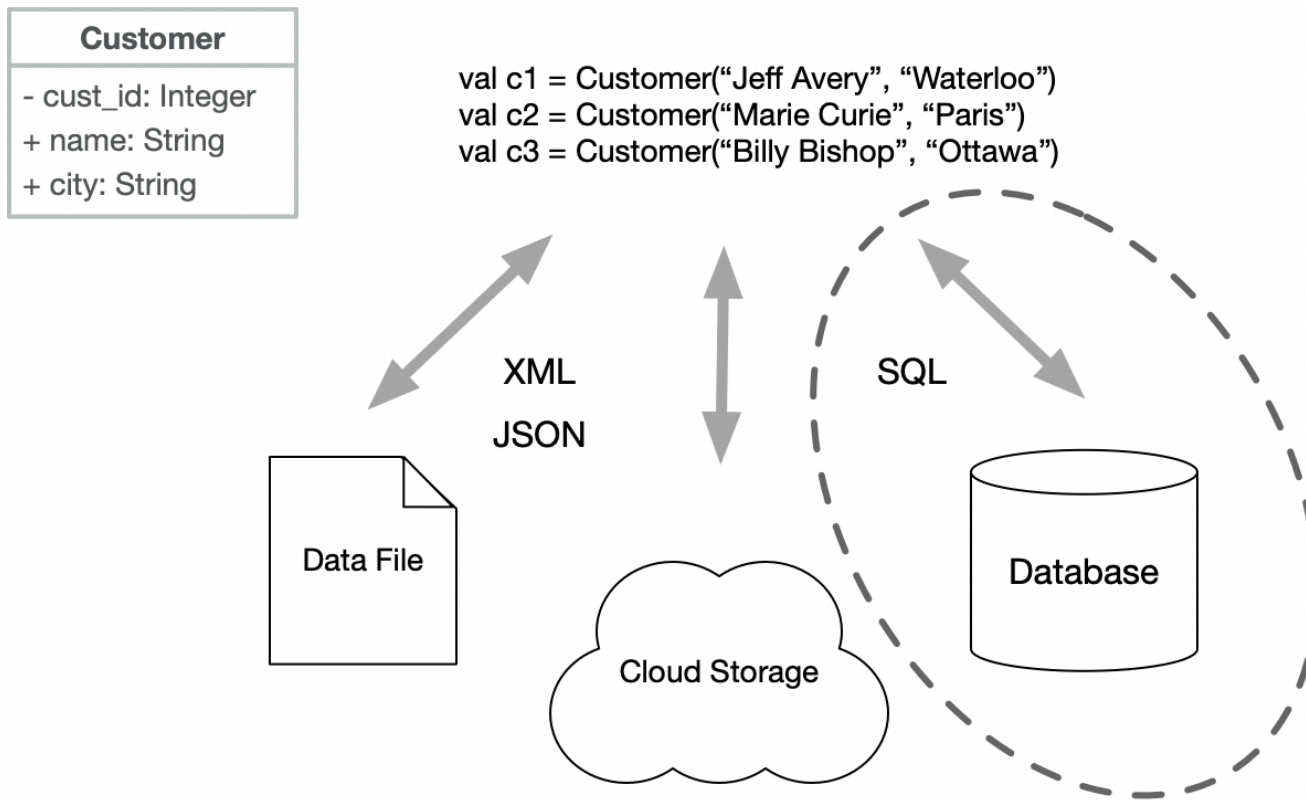
Databases range in size and complexity from simple file-based databases that run on your local computer, to large scalable databases. They are optimized for fetching text and numeric data, and performing set operations on this data.

They also have the advantage of scaling extremely well. They're optimized not just for efficient storage and retrieval of large amounts of data, but also for concurrent access by hundreds or thousands of users simultaneously<sup>1</sup>.

— — — — —

1. This is a **huge** topic and we're not even scratching the surface. I'd strongly encourage you to take a formal database course e.g. CS 348.

We're going to focus on one particular type: **relational databases**.



Kotlin uses a SQL library to query a relational database.

# Relational Databases

A relational database structures data into tables<sup>1</sup>:

- A **table** represents some logical entity e.g. Customer, Transactions. It consists of columns and rows.
  - A **row** is a **record**, containing values for each field. e.g. a Customer table would have one row for each of its customers.
  - A **column** is a **field** in that table. e.g. a “Customer“ table might have “name“, “city“, “birthdate“ fields in row, which are the fields for each customer.

— — —

1. The relational data model was proposed by E. F. Codd in 1970. Relational databases, that utilize this model, are common. e.g. Oracle, SQL Server, PostgreSQL, MySQL.

Here's our earlier example shown as a class, a set of records in a CSV file and as a set of relational database tables.

### Class

Customer
- cust_id: Integer
+ name: String
+ city: String

### CSV File

```
# cust_id, name, city
1001, Jeff Avery, Waterloo
1002, Marie Curie, Paris
1003, Billy Bishop, Ottawa
```

### Database Table

cust_id	name	city
1001	Jeff Avery	Waterloo
1002	Marie Curie	Paris
1003	Billy Bishop	Ottawa

### Transactions

- cust_id: Integer
- tx_id: Integer
+ item: String
+ amount: Double

```
# cust_id, tx_id, item, amount
1002, 43222, Chemistry set, 100.00
1003, 54187, Duct tape, 5.99
```

tx_id	cust_id	Item	Amount
43222	1002	Chemistry set	125.00
54187	1003	Duct tape	5.99

Data integrity is retained across data representations, even if the structure changes slightly

Why is this approach valuable?

1. Relational databases allow for very efficient storage and access of data.
2. Relational databases support operations on **sets** of records. e.g.
  - Fetch a list of all purchases greater than \$100.
  - Delete customers from "Ottawa".

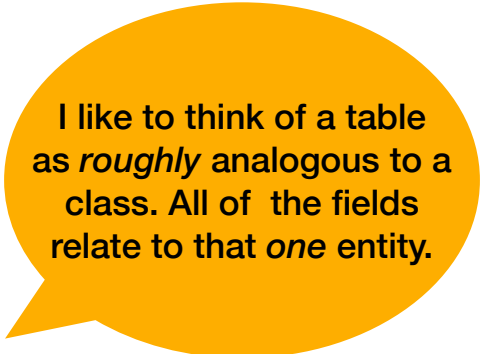
Our example is pretty trivial, but imagine *useful* queries like:

- "Find all transactions between 2:00 and 2:30", or
- "Find out which salesperson sold the most during Saturday's sale".

# Table

- A **table** is the foundational concept. It collects together a number of related **fields** into **records**.
- e.g. Customer table contains customer information
  - One record (row) per customer
  - One field (column) for each property of that customer.

cust_id	name	city
1001	Jeff Avery	Waterloo
1002	Marie Curie	Paris
1003	Billy Bishop	Ottawa



I like to think of a table as *roughly* analogous to a class. All of the fields relate to that *one* entity.

# Keys

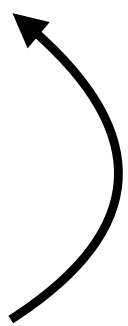
- A key is a column that helps us identify a row or set of rows in a table.
- A **primary key** is a column in a database with a value that *uniquely* identifies that row. A table cannot have more than one primary key.
- e.g. cust\_id is a unique identifier for each row in the customer table.
- Using “cust\_id=1002” to filter any operation will force that operation to only affect the “Marie Curie” record.

cust_id	name	city
1001	Jeff Avery	Waterloo
1002	Marie Curie	Paris
1003	Billy Bishop	Ottawa





# Keys

- A **foreign key** is a key used to refer to data being held in a different table.
  - Customer table
    - Primary key: cust\_id
  - Transactions table
    - Primary key: tx\_id
    - Foreign key: cust\_id
- 

Customer

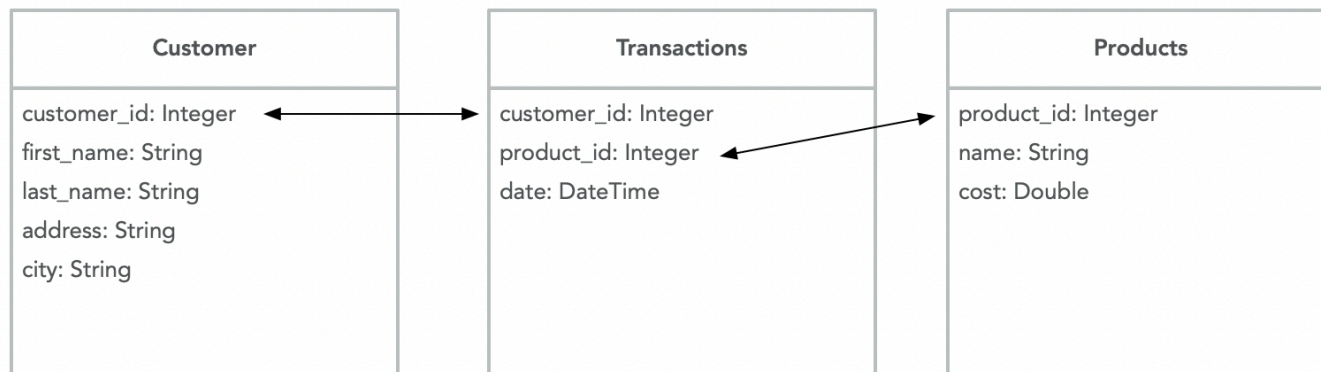
cust_id	name	city
1001	Jeff Avery	Waterloo
1002	Marie Curie	Paris
1003	Billy Bishop	Ottawa

Transactions

tx_id	cust_id	Item	Amount
43222	1002	Chemistry set	125.00
54187	1003	Duct tape	5.99

# Joins

It's common to have a record spread across multiple tables. A join describes how to relate data across tables using foreign keys.



```
customer_id: 1001 → customer_id: 1001 → product_id: 55
first_name: Jeff      product_id: 55 → name: T-shirt
last_name: Avery     date: 12-Aug-2020 cost: $29.95
address: 200 Main St.
city: Waterloo
```

**Record:** 1001, Jeff Avery, 200 Main St. Waterloo, T-shirt, \$29.95, 12-Aug 2020

Transactions: we need to join data to recreate a record.

# SQLite

# SQLite

SQLite (pronounced ESS-QUE-ELL-ITE) is a small-scale relational DBMS. It is small enough for local, standalone use and is preinstalled on Android and many operating systems.

"SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine.

SQLite is the most used database engine in the world.  
SQLite is built into all mobile phones and most computers..."

<https://www.sqlite.org/index.html>

# Installing SQLite

You can install the SQLite database under Mac, Windows or Linux.

1. Visit the [SQLite Download Page](#).
2. Download the binary for your platform.
3. To test it, launch it from a shell.

```
$ sqlite3
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .exit
$
```

4. We will create a database from the command line. Optionally, you can install [SQLite Studio](#), a GUI for managing databases.

# SQLite Demo

You can download the [SQLite Sample Database](#) and confirm that SQLite is working. We can open and read from this database using the command-line tools:

```
$ sqlite3
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database. sqlite>
```

SQLite database commands start with a period (.):

```
sqlite> .help
.help
.auth ON|OFF          Show authorizer callbacks
.backup ?DB? FILE     Backup DB (default "main") to FILE
....
```

# SQLite Commands

These commands are "meta-commands" that act on the database itself, and not the data that it contains. Some particularly useful commands:

Command	Purpose
<code>.open &lt;filename&gt;</code>	Open database <filename>.
<code>.database</code>	Show all connected databases.
<code>.log &lt;filename&gt;</code>	Write console to log <filename>.
<code>.read &lt;filename&gt;</code>	Read input from <filename>.
<code>.tables</code>	Show a list of tables in the open database.
<code>.schema &lt;table&gt;</code>	SQL to create a particular <table>.
<code>.fullschema</code>	SQL to create the entire database structure.
<code>.quit</code>	Quit and close connections.

We often use these meta-commands to change settings, and setup our queries.

```
$ sqlite3
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
```

```
sqlite> .open chinook.db // name of the file
sqlite> .mode column // lines up data in columns
sqlite> .headers on // shows column names at the top
```

```
// determine which tables to query
```

```
sqlite> .tables
albums      employees   invoices    playlists
artists     genres      media_types tracks
customers   invoice_items playlist_track
```



# SQL Queries

Once we've identified what we want to do, we can just execute our queries. Examples of selecting from a single table at a time:

```
sqlite> SELECT * FROM albums WHERE albumid < 5;
```

AlbumId	Title	ArtistId
1	For Those About To Rock	1
2	Balls to the Wall	2
3	Restless and Wild	2
4	Let There Be Rock	1

```
sqlite> SELECT * FROM artists WHERE ArtistId = 1;
```

ArtistId	Name
1	AC/DC

Example of a JOIN across two tables (based on a primary key, "ArtistId"). You often will have multiple WHERE clauses to join between multiple tables.

```
sqlite> SELECT albums.AlbumId, artists.Name, albums.Title
        FROM albums, artists
        WHERE albums.ArtistId = artists.ArtistId
        AND albums.AlbumId < 4;
```

AlbumId	Name	Title
1	AC/DC	For Those About To Rock
2	Accept	Balls to the Wall
3	Accept	Restless and Wild
4	AC/DC	Let There Be Rock

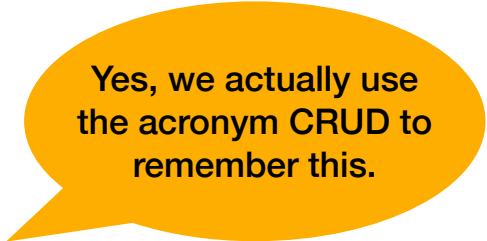
# Structured Query Language (SQL)

# What is SQL?

**SQL** (pronounced "Ess-que-ell") is a Domain-Specific Language (DSL) for describing your queries. Using SQL, you write statements describing the operation to perform, against which tables, and the database performs the operations for you.

SQL is a standard<sup>1</sup>, so SQL commands work across different databases. You can:

- **C**reate new records
- **R**etrieve sets of existing records
- **U**pdate the fields in one or more records
- **D**elete one or more records



Yes, we actually use the acronym **CRUD** to remember this.

— — —

1. SQL was adopted as a standard by ANSI in 1986 as SQL-86, and by ISO in 1987.

# Syntax

SQL has a particular syntax for managing sets of records:

```
<operation> FROM [table] [WHERE [condition]]
```

```
operations: SELECT, UPDATE, INSERT, DELETE, ...
```

```
conditions: [col] <operator> <value>
```

You issue English-like sentences describing what you intend to do.

SQL is **declarative**: you describe what you want done, but don't need to tell the database how to do it.

There's also a relatively small number of operations to support.

## Create: add new records

INSERT adds new records to your database.

```
INSERT INTO Customer(cust_id, name, city)
VALUES ("1005", "Sandra Avery", "Kitchener")
```

```
INSERT INTO Customer(cust_id, name, city)
VALUES ("1005", "Austin Avery", "Kitchener") // uh oh
```

## Retrieve: display existing records

SELECT returns data from a table, or a set of tables.

- asterix (\*) is a wildcard meaning “all”.
- an optional WHERE clause can filter the data.

```
SELECT * FROM customers
```

```
--> returns ALL data
```

```
SELECT * FROM Customers WHERE city = "Ottawa"
```

```
-- > {"cust_id":1003, "name":"Billy Bishop", "city":"Ottawa"}
```

```
SELECT name FROM Customers WHERE custid = 1001
```

```
--> "Jeff Avery"
```

## Update: modify existing records

UPDATE modifies one or more fields based **in every row that matches the criteria that you provide.**

If you want to operate on a single row, you need to use a WHERE clause to give it some criteria that makes that row unique.

```
UPDATE Customer SET city = "Kitchener" WHERE cust_id = 1001
```

-> updates one matching record since cust\_id is unique for each row

```
UPDATE Customer SET city = "Kitchener" // uh oh
```

-> no "where" clause, so we update everything to Kitchener.



## Delete: delete records

DELETE removes **every record from a table that matches the criteria that you provide.**

If you want to operate on a single row, you need to use a WHERE clause to give it some criteria that makes that row unique.

```
DELETE FROM Customer WHERE cust_id = 1001
```

-> deletes one matching record since cust\_id is unique for each row

```
DELETE FROM Customer// uh oh
```

-> deletes everything from this table

# WHERE clause

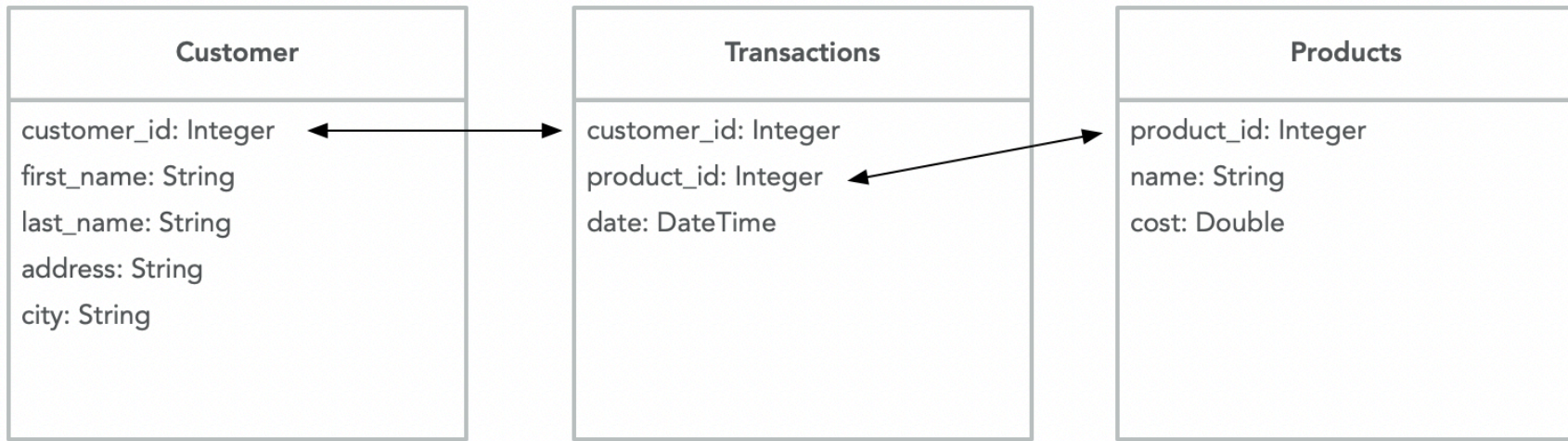
- A where clause allows us to filter a set of records.
- We've already seen this a few times:

```
UPDATE Customer SET city = "Kitchener" WHERE cust_id = 1001
```

-> updates one matching record since cust\_id is unique for each row

It also allows us to define relations between tables.

This means that we can start to run more complex queries across multiple tables.

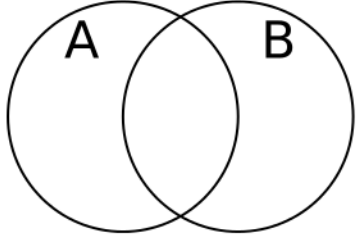
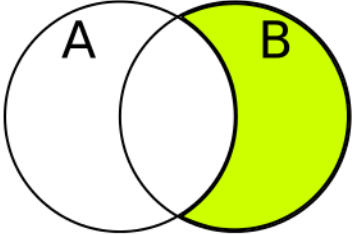
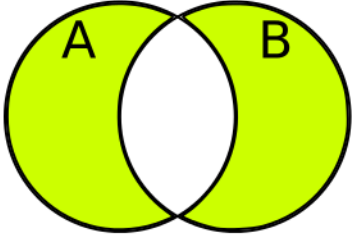
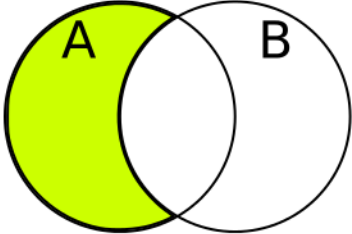
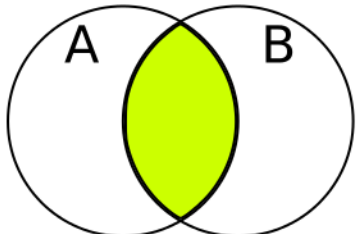
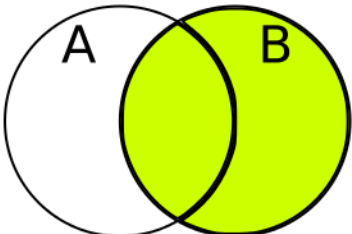
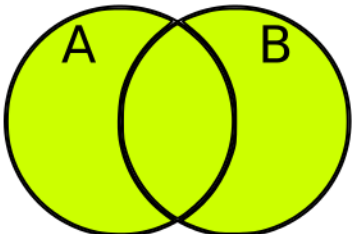
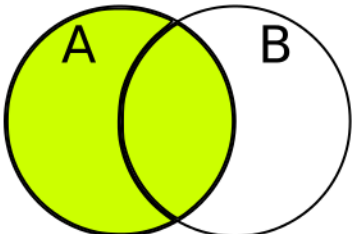


```
$ SELECT c.customer_id, c.first_name + " " + c.last_name, t.date,  
        p.name, p.cost  
FROM Customer c, Transactions t, Products p  
WHERE c.customer_id = t.customer_id  
AND t.product_id = p.product_id
```

```
$ 1001, Jeff Avery, 12-Aug-2020, T-shirt, 29.95
```

# SQL JOINS

Arranged in a Karnaugh Map by Jason Charney (jrcharney@gmail.com)

No join ( $\emptyset$ )	[Exclusive] Right Join ( $\neg A$ )	[Exclusive] Full Join ( $A \oplus B$ )	[Exclusive] Left Join ( $\neg B$ )
	 <pre>SELECT * FROM A RIGHT JOIN B ON A.key = B.key WHERE B.key IS NULL</pre>	 <pre>SELECT * FROM A FULL JOIN B ON A.key = B.key WHERE B.key IS NULL OR A.key IS NULL</pre>	 <pre>SELECT * FROM A LEFT JOIN B ON A.key = B.key WHERE B.key IS NULL</pre>
Inner Join ( $A \wedge B$ )	[Inclusive] Right Join ( $B$ )	[Inclusive] Full Join ( $A \vee B$ )	[Inclusive] Left Join ( $A$ )
 <pre>SELECT * FROM A INNER JOIN B ON A.key = B.key</pre>	 <pre>SELECT * FROM A RIGHT JOIN B ON A.key = B.key</pre>	 <pre>SELECT * FROM A FULL JOIN B ON A.key = B.key</pre>	 <pre>SELECT * FROM A LEFT JOIN B ON A.key = B.key</pre>

Our examples are all Inner Joins, using a simpler syntax.

# Accessing Database in Code

# Accessing Databases in Code

Kotlin doesn't have built-in database support, but it does include a library for this.

Kotlin leverages the **Java JDBC API** ("Java DataBase Connectivity"). This provides a standard mechanism for connecting to databases, issuing queries, and managing results.

Creating a database project in IntelliJ:

1. Create a Gradle/Kotlin project.
2. Modify the "build.gradle" to include a dependency on SQLite/JDBC.

```
implementation 'org.xerial:sqlite-jdbc:3.30.1'
```

3. Use the Java SQL package classes to connect and fetch data.

```
java.sql.Connection  
java.sql.DriverManager  
java.sql.SQLException
```

This example uses a **sample database** from the SQLite tutorial.

```
fun connect(): Connection? {
    var conn: Connection? = null
    try {
        val url = "jdbc:sqlite:chinook.db"
        conn = DriverManager.getConnection(url)
        println("Connection to SQLite has been established.")
    } catch (e: SQLException) {
        println(e.message)
    }
    return conn
}
```

Public repository: /databases/JDBC

```

fun query(conn:Connection?) {
    try {
        if (conn != null) {
            val sql = "select albumid, title, artistid from albums where albumid < 5"
            val query = conn.createStatement()
            val results = query.executeQuery(sql)
            println("Fetched data:");
            while (results.next()) {
                val albumId = results.getInt("albumid")
                val title = results.getString("title")
                val artistId = results.getInt("artistid")
                println(albumId.toString() + "\t" + title + "\t" + artistId)
            }
        }
    } catch (ex: SQLException) {
        println(ex.message)
    }
}

```

```

Connection to SQLite has been established.
Fetched data:
1           For Those About To Rock We Salute You.      1
2           Balls to the Wall                          2
3           Restless and Wild                          2
4           Let There Be Rock                          1
Connection closed.

```



# Tips on Using SQLite

- The connection string (`val url = "jdbc:sqlite:chinook.db"`) describes the database name, including the path to the database.
- The sample project doesn't include a path, which means that it looks for the database in the current running directory. This is why the sample database is located in the root of the project.
- You will want your application to access an instance of the database when it runs. You can either:
  1. **Ship with an empty database.** You can pre-populate it with the tables and data that you need. If you do this, you might need a way to create a “clean” database every time that you run your app.
  2. **Create a new empty database when you launch your app.** If your database does not exist when you attempt to connect to it, it will be created for you. However, it will be empty: no tables, or data. For this to be useful, your app would need to create tables and starting data when it's launched.