Databases

CS 346 Application Development

The need for data management

- Most applications that you build will have data that they need to persist. e.g.,
  - a music player might store playlists of music,
  - a social media application might store account and login information,
  - a photo editor might store your preferred file location, and image settings,
  - any application might store preferred theme, window size and location.

- You will typically store this type of data in some persistent location:
  - e.g., file on the local filesystem, or a local or remote database.

What is a database?

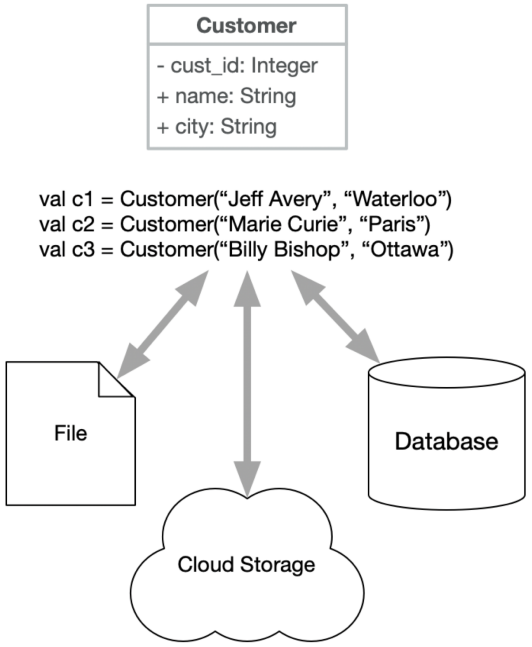
A database is a system for **storing data as records.**

- It's designed to handle **large volumes of data**.
- It supports **efficient and complex operations**.
- Facilitates **data sharing** across concurrent users/systems (which makes it immediately superior to raw data files).

A **record** is a set of associated **fields**.

- A `Customer class` describes a Customer as a set of properties and behaviours.
- A `Customer record` describes it as a set of fields.

```
e.g.,
Customer(cust_id=1001, name="Jeff Avery", city="Waterloo")
```



Applications often consume data from many different sources and need to store it in a consistent manner.

Data guarantees: ACID

ACID represents a set of properties intended to guarantee data validity despite errors, power failures, and other mishaps.

Ideally, we want these properties in our database:

- **Atomicity** prevents updates to the database from occurring only partially.
- **Consistency** guarantees that transaction can move database from one valid state to the next. All associated changes take place together.
- **Isolation** determines how a particular action is shown to other concurrent system users. e.g., how you handle cases of reading data that is being changed.
- **Durability** is the property that guarantees that transactions that have been committed will survive permanently.

Relational Databases

There are many different types of databases. The two main approaches that you will encounter are:

- **Relational** (SQL) databases for structured data e.g., Oracle, PostgreSQL, MySQL
- **Document** (NoSQL) databases for unstructured data e.g., MongoDB

We'll focus on relational databases:

1. ACID compliance. NoSQL databases often trade safety for performance.
2. They allow for very efficient data storage with little redundancy.
3. They are optimized for operations on sets of records.
   - This mirrors how we want to work with our data.
   - e.g., "fetch all sales that were recorded last night in the Chicago office".

Relational Databases

Old-school, SQL databases. Still incredibly useful.

Table

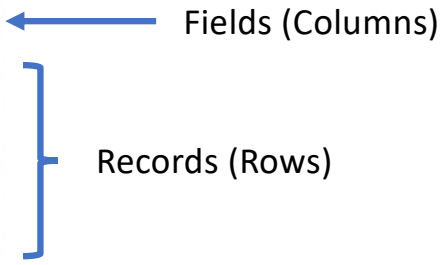
A **relational database** structures data into **tables** representing logical entities e.g. `Customer` and `Transaction` tables. Records are stored as rows in each table.

A **table** is a foundational concept. It collects 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



Data -> 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

Primary Key

You need a way to uniquely identify each record in a table.

- 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 each row. A table cannot normally have more than one primary key.
  - In the table below, `cust_id` is a unique identifier for each row in the customer table.
  - Specifying `cust_id=1002` will restrict an operation to the `Marie Curie` record.

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

Records spanning tables

Relational databases reduce data redundancy by splitting records across multiple tables.

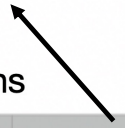
e.g.

Imagine that we have an online store. We want to track both `Customer` and `Transaction (Sales)` information, so we split this data across two tables.

If Mme. Curie purchases something else later, we only need to add one Transaction row; the Customer row doesn't change.

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

Using foreign Keys

How do we associate our two tables?

A **foreign key** is a key used to refer to data being held in a different table.

A primary key of one table is the foreign key 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

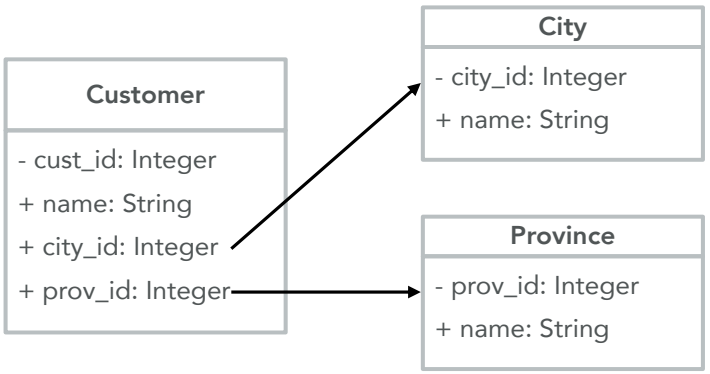
tx_id	cust_id	Item	Amount
43222	1002	Chemistry set	125.00
54187	1003	Duct tape	5.99

Each transaction can be uniquely identified by the primary key **tx_id**. It is linked to a unique customer through **cust_id**.

Reconstructing a record

We want to store our data using multiple tables to avoid redundancy, but we will eventually want to recreate the complete record.

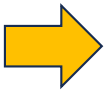
101, **Jeff Avery**, 233, 5
102, Brien Bendis, 231, 3
103 Julie Zhang, 155, 2



231, Wallaceburg
232, Washago
233, **Waterloo**
…

3, Saskatchewan
4, Manitoba
5, **Ontario**
…

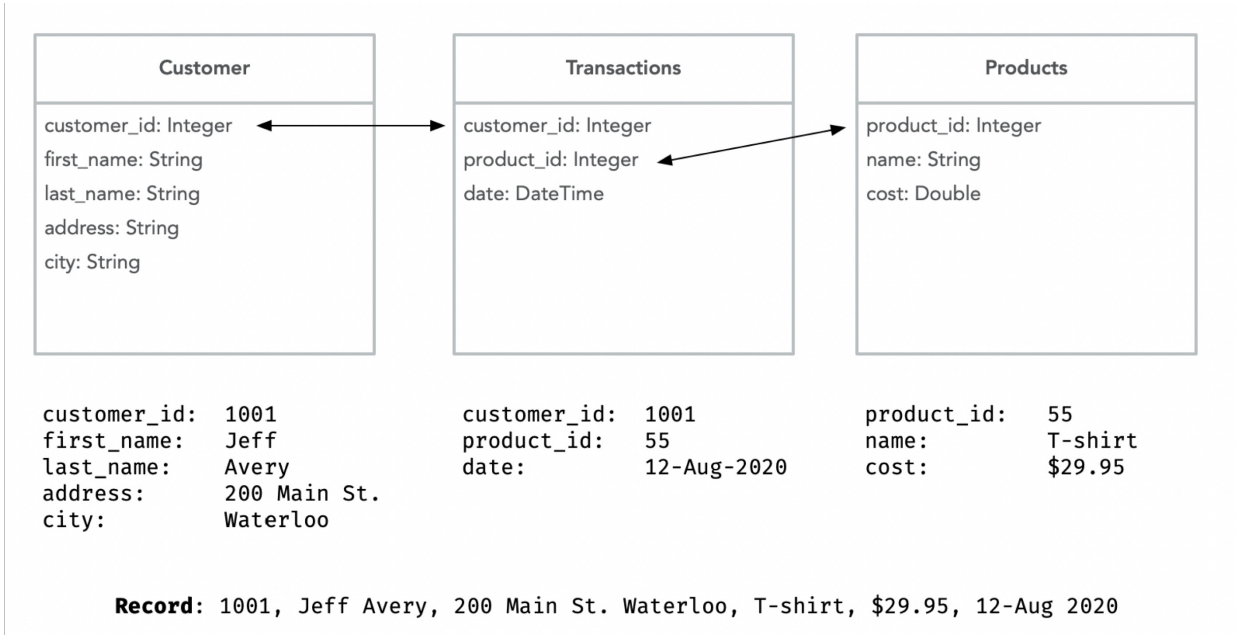
101, **Jeff Avery**, 233, 5
233, **Waterloo**
5, **Ontario**,



101, Jeff Avery, Waterloo, ON

Joins

A `join` describes the relationship between records in different tables. We split data for efficiency but joins lets us reassemble records when we need that information again. *We'll revisit joins in a moment…*



Transactions ensure ACID

To achieve safety, we treat multiple actions that are being performed as a single unit of work called a **transaction**.

- All changes are performed together (atomic).
- If there is any error in performing an action, we undo *all* of these actions.

- How do you use this?
  - "start" a transaction when you perform a series of operations and
  - "commit" when you are done.

Details will vary by database

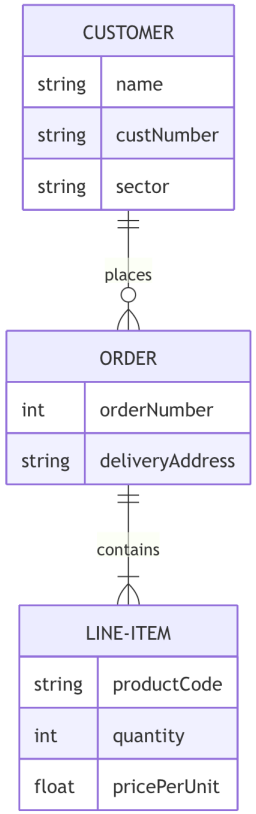
- This is how we can handle:
  - Two or more users are updating the same data at the same time, or
  - One person reading data while one modifies it, or
  - An update failure that doesn't leave data in an inconsistent state.

Designing a database

How do you decide on the structure? How do you create and manage it?

Steps to create a database

1. Decide on the tables (entities) to model.
    - These should come from your requirements.
    - What data do you need to store for your features?
    - Optional: Entity-Relationship Diagram (ERD)
2. Normalize the schema to remove inefficiencies.
    - 1NF, 2NF, 3NF
3. Design the tables.
    - Columns, data types, relationships, keys.

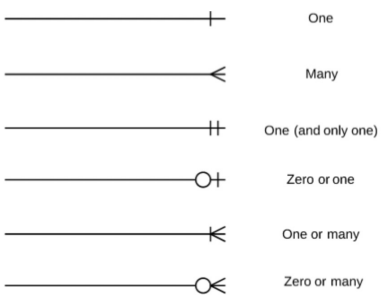


An Entity Relationship Diagram (ERD) created in Mermaid.

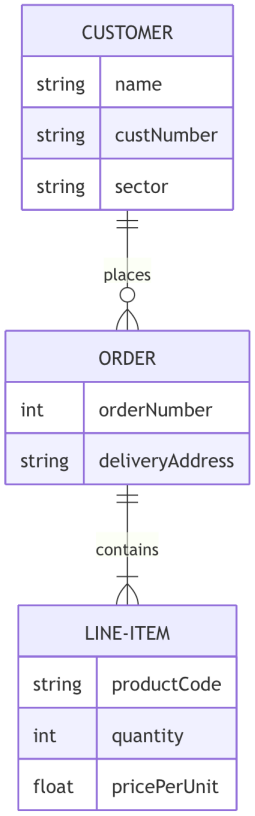
What is an ERD?

A diagram that shows entities and their relationships.

- **Tables**: entities that we wish to model.
  - e.g., Customer, Order and Line-Item.
- **Columns**: fields in each entity.
  - e.g., name, custNumber, sector.
- **Relationships**: how entities are related.
  - e.g., customer-places-order
  - e.g., order-contains-line-item.
- **Cardinality**: numerical relationship between rows of one table and rows in another.
  - e.g., 1 customer places 0 or more orders.
  - e.g., 1 order contains at least one line item.



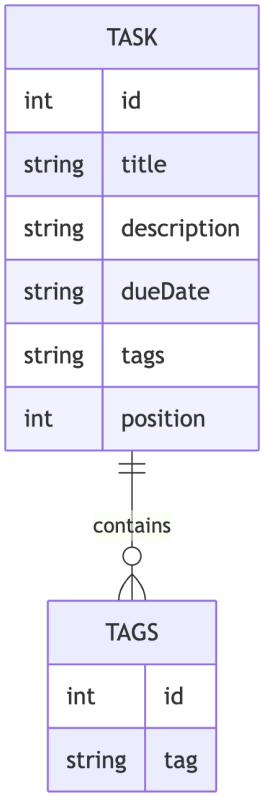
Cardinality symbols



See mermaid.js to generate diagrams.

Design Tables

- Start by thinking about your requirements.
  - Which entities do you need to represent your data?
  - Entities → Tables
- If you're unsure of where to start, make a list of all the data items that you need to satisfy your functional requirements.
  - e.g., `mm-android` application had entities for TASK and TAGS.
  - 1 task contains 0 or more tags (strings associated with a task).
- If you've already got your application working and are adding the database now? Look at your data classes for ideas.
  - Each data class is probably an entity.



Normalization

Constraint (informal description in parentheses)	UNF (1970)	1NF (1970)	2NF (1971)	3NF (1971)
Unique rows (no duplicate records)[4]	✓	✓	✓	✓
Scalar columns (columns cannot contain relations or composite values)[5]	✗	✓	✓	✓
Every non-prime attribute has a full functional dependency on each candidate key (attributes depend on the *whole* of every key)[5]	✗	✗	✓	✓
Every non-trivial functional dependency either begins with a superkey or ends with a prime attribute (attributes depend *only* on candidate keys)[5]	✗	✗	✗	✓

1NF: each field is atomic, containing a single value rather than a set of values

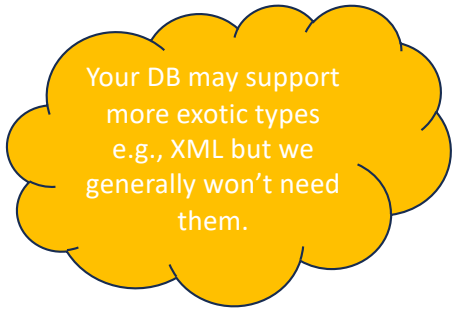
sufficient for us

2NF: 1NF and no partial dependencies i.e., partial composite keys.

3NF: 2NF and each dependency must depend solely and non-transitively on the candidate key.

-- Wikipedia

Establish Column Data Types



- Numeric:
  - INTEGER: whole numbers
  - DECIMAL: fixed precision and scale e.g., monetary values
  - FLOAT: floating point
- Character
  - CHAR: fixed length strings
  - VARCHAR: variable length strings up to a max length
  - TEXT: large blocks of text e.g., text editor
- Date
  - DATE: date in YMD
  - TIME: time in HMS

Determine Relationships (and Keys)

- How are your tables related to one another?
- Each table needs a primary key
  - Should be auto-generated by the database
  - Use constraints e.g., NOT NULL to enforce data integrity
- Allow the libraries to enforce relationships
  - Specify PK and FK relationships in your schema
  - Don’t rely on yourself to form queries properly (it’s easy to make mistakes!)

Structured Query Language (SQL)

How to write queries for your relational database.

How do we perform database operations?

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 and the database performs them for you.

- SQL is an ANSI/ISO standard[1], so SQL commands work the same way across different relational databases. You can use it to:
  - **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

SQL 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, “Molly Malone", "Kitchener")
```

```
INSERT INTO Customer(cust_id, name, city)
 VALUES (1005, “April Ludgate", "Kitchener") // problem?
```

Retrieve: Display existing records

SELECT returns data from a table, or a set of tables. NOTE: Asterix (*) is a wildcard meaning “all”.

```
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 record since cust_id is unique for each row

UPDATE Customer SET city = "Kitchener" // uh oh
-> no “where” clause, so we change all records to Kitchener.
```

Delete: Remove 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
```

```
DELETE FROM Customer// uh oh
-> deletes everything from this table
```

Filtering with a “where” clause

A `where` clause allows us to filter a set of records.

e.g.,

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

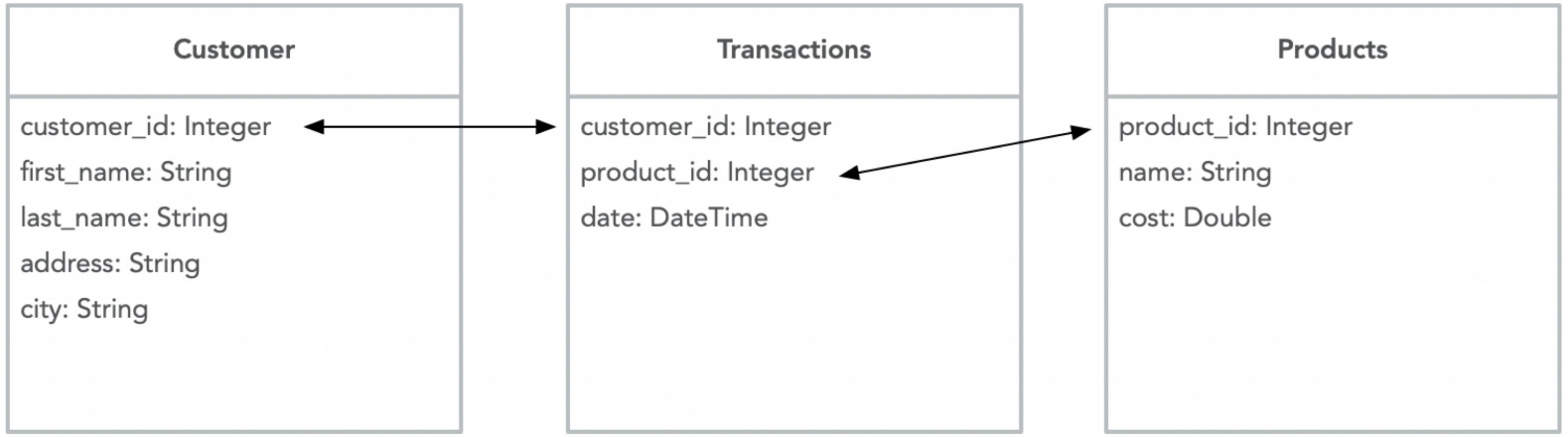
Sorting with an “order by” clause

An “order by” sorts the results by the column name that you specify.

e.g.,

```
SELECT * FROM Products ORDER BY Price;
```

Joining records




```
SELECT c.customer_id, c.first_name + " " + c.last_name, t.date, p.name, p.cost
 FROM (Customer c
 INNER JOIN Transactions t ON c.customer_id = t.customer_id)
 INNER JOIN Products p ON t.product_id = p.product_id)
```

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

Types of SQL joins

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

These two tables are related through the CustomerID column.

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujilo Emparedados y helados	Ana Trujilo	Mexico
3	Antonio Moreno Taqueria	Antonio Moreno	Mexico

We have multiple ways that we can associate these tables based on this relationship.

Example from https://www.w3schools.com/sql/sql_join.asp

Types of SQL joins

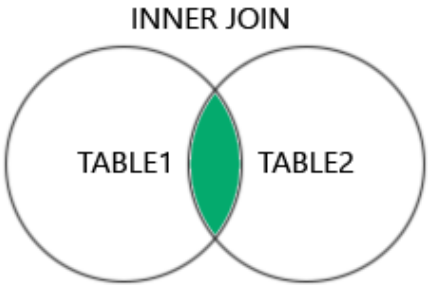
- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table

Example: INNER Join

```
SELECT ProductID, ProductName, CategoryName
FROM Products
INNER JOIN Categories
ON Products.CategoryID = Categories.CategoryID;
```

Only returns values where the CategoryID exists in both Products and Categories.

e.g., if a Product existed in the Product table but there was no corresponding category in the Categories table, then it would now show up in the query results.

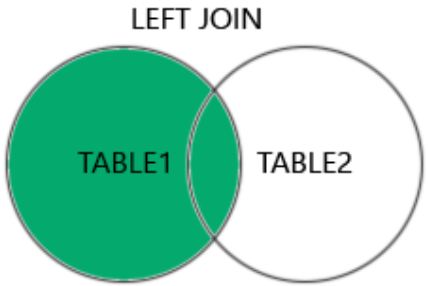


Example: LEFT (OUTER) Join

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Returns all the records from the left-most table, and the matching records from the right-hand table. If there is no match in the right-hand side, those fields will be left blank.

e.g., if a Product existed in the Product table but there was no corresponding category in the Categories table, then it would show up in the query results with an empty category.

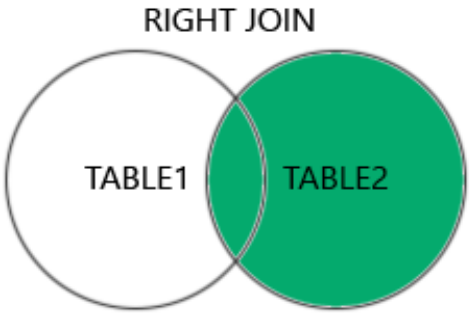


Example: RIGHT (OUTER) Join

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Returns all the records from the right-most table, and the matching records from the left-hand table. If there is no match in the left-hand side, those fields will be left blank.

e.g., in our product example where there were no matching categories, you would retrieve all the categories but non-matching products would be blank.



SQLite

Using a simple library-based relational database.

Introduction

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

Things to know about SQLite

- SQLite is a C-library, that can be installed practically anywhere.
  - It's preinstalled on macOS, Linux and Android.
- Your database is stored on a local file-system.
  - Should be accessible to your application.
  - Your database is actually a single file e.g., `chinook.db`.
- It's intended for single-user use only.
  - No authentication i.e. no username/password required.
  - Secure the database the same way that you would a file e.g., encrypted on a local hard drive, in the user's home directory.
- It's *blisteringly* fast and very lightweight.

Check the installation

Run the `sqlite3` command to see if it's preinstalled.

You can also download and open a sample database e.g., `chinook.db`

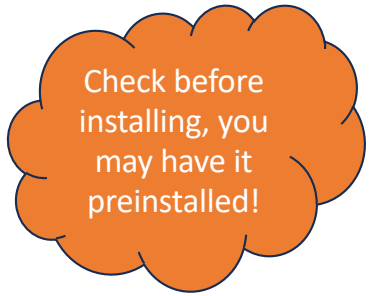
```
$ sqlite3
SQLite version 3.43.2 2023-10-10 13:08:14
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.

sqlite> .open chinook.db
sqlite> .tables
albums          employees       invoices        playlists
artists         genres          media_types     tracks
customers       invoice_items   playlist_track
sqlite> select * from artists;
…
```

Installation

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

1. Visit the SQLite Download Page. Download the binary for your platform.
2. 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
```

Useful Commands

To get a list of commands, run `sqlite3` and then enter `.help.

Command	Purpose
.open <filename>	Open database <filename>.
.database	Show all connected databases.
.log <filename>	Write console to log <filename>.
.read <filename>	Read input from <filename>.
.tables	Show a list of tables in the open database.
.schema <table>	SQL to display create stmt for a <table>.
.fullschema	SQL to create the entire database structure.
.quit	Quit and close connections.

```
$ sqlite3
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.

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


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

sqlite> .schema genres
CREATE TABLE IF NOT EXISTS "genres"
(
    [GenreId] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    [Name] NVARCHAR(120)
);
```

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
```

Accessing a database

Options for connecting your application to a relational database.

Using JDBC to connect

Kotlin can use the **Java JDBC API** to connect to any compliant database, including SQLite.

- Most/all databases have a JDBC driver available.
- Google database + “JDBC” to locate drivers.

To create a database project in IntelliJ:

1. Create a Gradle/Kotlin project.
2. Modify the build.gradle.kts to include your database driver:
   ```
   implementation(“org.xerial:sqlite-jdbc:3.50.3.0”) // see GitHub
   ```
3. Use the Java SQL package classes to connect and fetch data.

Creating a connection

This example uses a sample database from the SQLite tutorial.

- We first create a connection to the database.
- The URL designates the type of database, and location of the database file.

```
fun connect(): Connection? {
    var connection: Connection? = null
    try {
        val url = "jdbc:sqlite:chinook.db” // URL format varies by driver
        connection = DriverManager.getConnection(url)
        println("Connection to SQLite has been established.")
    } catch (e: SQLException) {
        println(e.message)
    }
    return connection
}
```

Running a query

```
fun query(connection: Connection?) {
    try {
        if (connection != null) {
            val sql = "select albumid, title, artistid from albums where albumid < 5"
            val query = connection.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)
    }
}
```

```
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
```

Why you shouldn't use JDBC like this

JDBC is a useful mechanism for connecting to remote databases, but making raw SQL calls through the sql packages is error-prone:

- No type checking,
- No other safety mechanisms in-place.
- Requires us to explicitly convert between string data and class objects that are holding our data.

Not recommended, for any scenario.

A better-practice is to use a library that abstracts this functionality:

- Exposed is a JetBrains library for working with JDBC databases. It works with desktop but not Android.
- Room is a Google library for working with SQLite databases. It works with both desktop and Android.

Option 1: Exposed (JVM)

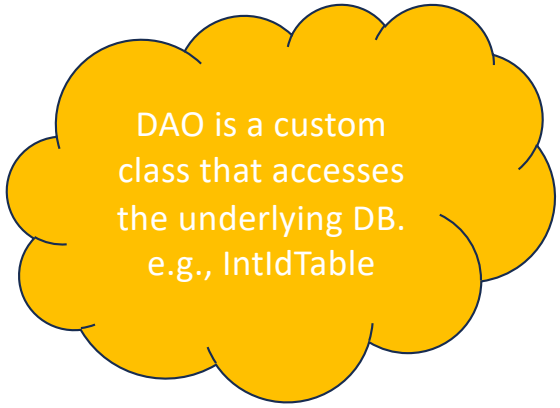
Exposed is a framework that provides a cleaner interface for working with JDBC. It provides two approaches:

- **Domain Specific Language** (DSL) – if you want a query language,
- **Data Access Objects** (DAO) – classes to abstract DB access.

Exposed works through JDBC and supports most popular databases including SQLite, H2, Oracle, Postgres…

- https://github.com/JetBrains/Exposed

Example: mm-desktop




```
class DBStorage(databaseName: String = ".mm.db"): IStorage {
  object TaskTable : IntIdTable() {
        val position = integer("position")
        val title = varchar("title", length = 256)
        val description = varchar("description", length = 256)
        val dueDate = varchar("due_date", length = 16)
        val tags = varchar("tags", length = 256)
  }


  init {
        Database.connect("jdbc:sqlite:$databaseName", "org.sqlite.JDBC")
        transaction {
                SchemaUtils.create(TaskTable, TaskTagTable, TagTable)
        }
  }
  // other methods
}
```

GitLab: demos > mm-desktop

Example: mm-desktop

```
override fun read(id: Int): Task? {
      return transaction {
          TaskTable
              .selectAll()
              .where { TaskTable.id eq id }
              .map { it ->
                  Task(
                      id = it[TaskTable.id].value,
                      position = it[TaskTable.position],
                      title = it[TaskTable.title],
                      description = it[TaskTable.description],
                      dueDate = it[TaskTable.dueDate],
                      tags = it[TaskTable.tags]
                  )
              }.firstOrNull()
      }
  }
```

GitLab: demos > mm-desktop

When to use Exposed?

- Exposed is ideal for back-end or JVM solutions.
  - e.g., a desktop application, or a web server accessing a database.
- Works great with coroutines, suspending functions.
- Highly recommended for web services that need DB access.

- It is NOT recommended for mobile development.
  - JDBC drivers are not usually intended to run on Android.
  - Android has its own (better performing) solution.

Option 2: Room (Android)

Google created Room in 2017, as an abstraction layer over SQLite.

Addresses runtime stability issues that you get when you work with low-level APIs:

- Compile time verification of SQL queries.
- Checks for missing tables and other entities to avoid runtime crashes.

It's also designed around the use of Data Access Objects (DAO).

- Room + SQLite is Google's recommended solution for Android.

@Entity

@Entity

- A Domain Object that reflects rows in a table (i.e., table structure).
- Effectively a **data class** with annotations for columns, keys.

```
@Entity
data class Contact(
  @PrimaryKey(autogenerate = true)
  val id: Int = 0,
  val firstName: String,
  val lastName: String,
  val phoneNumber: String
)
```

@ denotes an annotation. The compiler will replace these expressions with code.

@Dao

@Dao (Data access object)

- DAO that represents methods to access the database

```
@Dao
interface ContactDao {
  @Upsert
  suspend fun upsertContact(contact: Contact)

  @Delete
  suspend fun deleteContact(contact: Contact)

  @Query("SELECT * FROM contact ORDER BY firstName ASC")
  fun getContactsOrderedByFirstName(): Flow<List<Contact>>

  @Query("SELECT * FROM contact ORDER BY lastName ASC")
  fun getContactsOrderedByLastName(): Flow<List<Contact>>
}
```

suspend denotes a suspending function; think of it as a function that can suspend itself while waiting for the database function to complete. We'll discuss this more in the coroutines lecture.

@Database

@Database

- DAO that represents the main access point to the database.

```
@Database(entities = [Contact::class], version = 1)
abstract class ContactDatabase: RoomDatabase() {
    abstract val dao: ContactDao
}
```

Example: mm-android

MainActivity: binds things together

```
/*
 MainActivity launches everything
 The Application screen only accesses data through the ViewModel
 */
val database: TaskDb = getRoomDatabase(this)
val model = Model(database.taskDao())
val viewModel = TaskViewModel(model)

setContent {
    MMTheme {
        Application(viewModel)
    }
}
```

Example: mm-android

TaskEntity: models a table.

```
@Entity(tableName = "task_table")
data class Task(
    @PrimaryKey(autoGenerate = true)  val id: Int = 0,
    @ColumnInfo(name = "title")              val title: String,
    @ColumnInfo(name = "description")        val description: String,
    @ColumnInfo(name = "due_date")    val dueDate: String,
    @ColumnInfo(name = "tags")               val tags: String,
    @ColumnInfo(name = "position")    val position: Int,
)
```

This is just a data class with annotations added to it.

Example: mm-android

TaskDao: interactions with the database

```
@Dao
interface TaskDao: IDao {
    @Query("SELECT * FROM task_table")
    override fun getAll(): Flow<List<Task>>

    @Query("SELECT * FROM task_table WHERE id = :id")
    override suspend fun getById(id: Int): Task

    @Query("DELETE FROM task_table")
    override suspend fun deleteAll()

    @Delete
    override suspend fun delete(task: Task)

    @Insert
    override suspend fun insert(task: Task)

    @Update
    override suspend fun update(task: Task)
}
```

The Room annotation processor will generate code for these methods!

Example: mm-android

TaskDb: the database object

```
/*
 * Room DB
 * https://developer.android.com/training/data-storage/room
 */

@Database(entities = [Task::class], version = 1, exportSchema = false)
abstract class TaskDb : RoomDatabase(), IDb {
    abstract override fun taskDao(): TaskDao
}

fun getRoomDatabase(applicationContext: android.content.Context): TaskDb {
    val builder = databaseBuilder<TaskDb>(
        context = applicationContext,
        name = applicationContext.getDatabasePath("task.db").toString()
    )
    return builder
        .fallbackToDestructiveMigrationOnDowngrade(true)
        .setQueryCoroutineContext(Dispatchers.IO)
        .build()
}
```

Example: mm-android

Application: using the data in a View

```
@Composable
fun Application(viewModel: TaskViewModel) {
    val items by viewModel.getAll().collectAsState(initial = emptyList())

    LazyColumn(modifier = Modifier.fillMaxSize().padding(16.dp)) {
        items(items.size) { item ->
          TaskItem(
            task = items[item],
            isSelected = (items[item] == viewModel.selectedTask),
            onClick = { viewModel.selectedTask = items[item] },
            onDoubleClick = {
                viewModel.selectedTask = items[item]
                viewModel.showEditDialog = true
            }
        )
    }
}
```

Why you don’t want to use Room

- Room is SQLite only.
  - SQLite isn’t really meant for remote data. It’s great as a local solution, but you can’t easily host it online.

- If you need to share data, you want a large-scale solution
  - e.g., PostgreSQL, Oracle or something similar.

Option 3: Native SDK (Remote)

If you want to run against an online database you probably want a hosted solution i.e. a platform that provides access.

- Hosted platforms will typically provide an SDK/access method.

Online platforms that support Kotlin:

- Supabase has a Kotlin client library that you can use to access a Postgres database.
- Neon can be used with JDBC and also provides Postgres access.

*You are free to use other platforms that provide a **SQL database** and have adequate security in-place to restrict access.*

- *You do NOT want to put your data in an open and insecure database on the Internet.*

Testing

How to test your database.

Testing strategy

- Your database is an example of an external dependency
  - Your application accesses it, but it's effectively a "black box".
  - You cannot directly control how data is managed.
  - You cannot and should not test it directly.
- Alternatives to testing live data
  - You want to avoid testing against live data!
  - Alternatives
    - Mock data storage
    - Test database that mirrors your live database - challenge of getting the structure correct

GitHub: `demos` > `courses` uses Exposed and has DB unit tests.

Reference

- Google. 2025. Room for Kotlin Multplatform
- Lackner. 2025. The FULL Beginner Guide for Room in Android
- Muntenescu. 2021. Kotlin: Using Room Kotlin APIs - MAD Skills
- Nilanjan. 2023. How to Access Database with Kotlin JDBC
- SQLite. 2023. SQLite Documentation.
- Various. 2025. Chinook Database
- Various. 2025. MS Northwind Database for SQLite
- W3Schools. 2023. Introduction to SQL.