Databases

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

Motivation

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 (DBMS) is a system that manages and stores structured data [1].

- It's a standalone system that runs locally or "in the cloud".
- Handles user control, sharing, security, querying and managing data.

Databases store `records` which consist of `fields`. This is analogous to how we model data entities in code.

- A logical Customer record could be represented by an object containing properties and behaviours.
- A record in a database could consist of just the fields and their relations.

```
Customer(
  name="Jeff Avery",
  city="Waterloo"
)
```

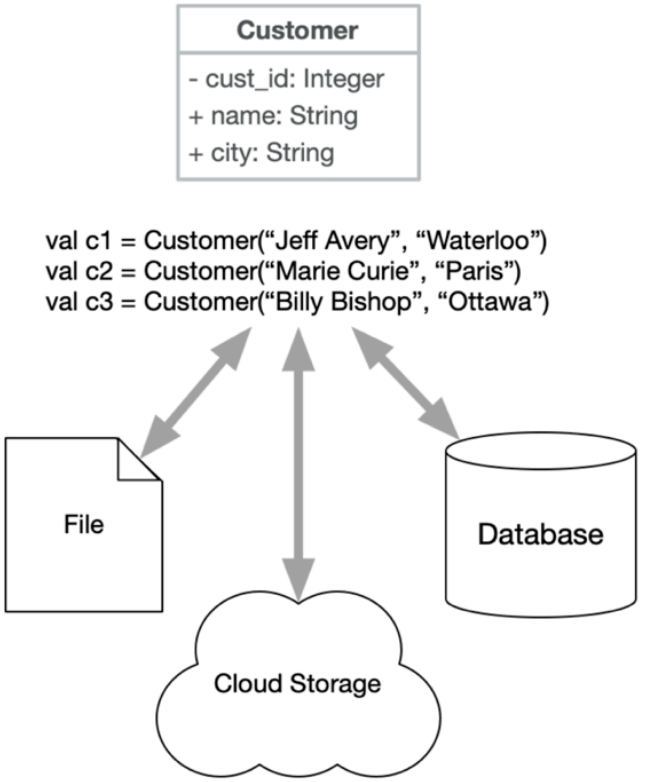

Figure 1: Applications can work with data from many sources: database, web or files.

Types of Databases

A database model refers to the logical structure of a database i.e., how data is represented and stored.

Logical data models include:
- Hierarchical model that gather information into tree-like structures.
- Relational model that uses a table-based format for storing recoreds.
- Network model which organizes data into a graph, where records are nodes.
- Document model that encode record data into a single self-contained document.

The most common databases used for application development are:
- Relational (SQL) e.g., Oracle, Postgres.
- Document (NoSQL) e.g., MongoDB, Google Cloud Firestore.

We'll focus on relational databases! They're flexible, high-performance and extremely popular.

Relational Databases

Concepts

Tables

A relational database structures data into `tables` containing `records`, which in turn contain `fields` holding the actual data. e.g. Customer table contains customer information

- One record (row) per customer
- One field (column) for each property of that customer record.

Customer

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

Concepts

Keys

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.

Primary Key

- 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 Customer 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.

Customer

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

Concepts

Foreign Keys

Logical records don't always fit into a single table. Sometimes, to remove redundancy, we split out data into separate tables e.g., if we have thousands of `Customers`, it's space inefficient to have a column for `City` that repeats "Waterloo" for every record.

Let's model an online store with both `Customer` and `Transaction` data. We'll split this across two tables, keeping our customer and transaction data separate. Notice that the `cust_id` is used to tie records together.

Customer

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

Transactions

tx_id	cust_id	item	price
43222	1002	Chem set	$125.00
54187	1003	Duct tape	$5.99

Concepts

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.
- This is the how we model relationships between tables.

Customer table

- Primary key: cust_id

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

Transactions table

- Primary key: tx_id
- Foreign key: cust_id

tx_id	cust_id	item	price
43222	1002	Chem set	$125.00
54187	1003	Duct tape	$5.99

Concepts

Joins

A join describes the relationship between records in different tables. We split data for efficiency but joins lets us reassemble records when needed.

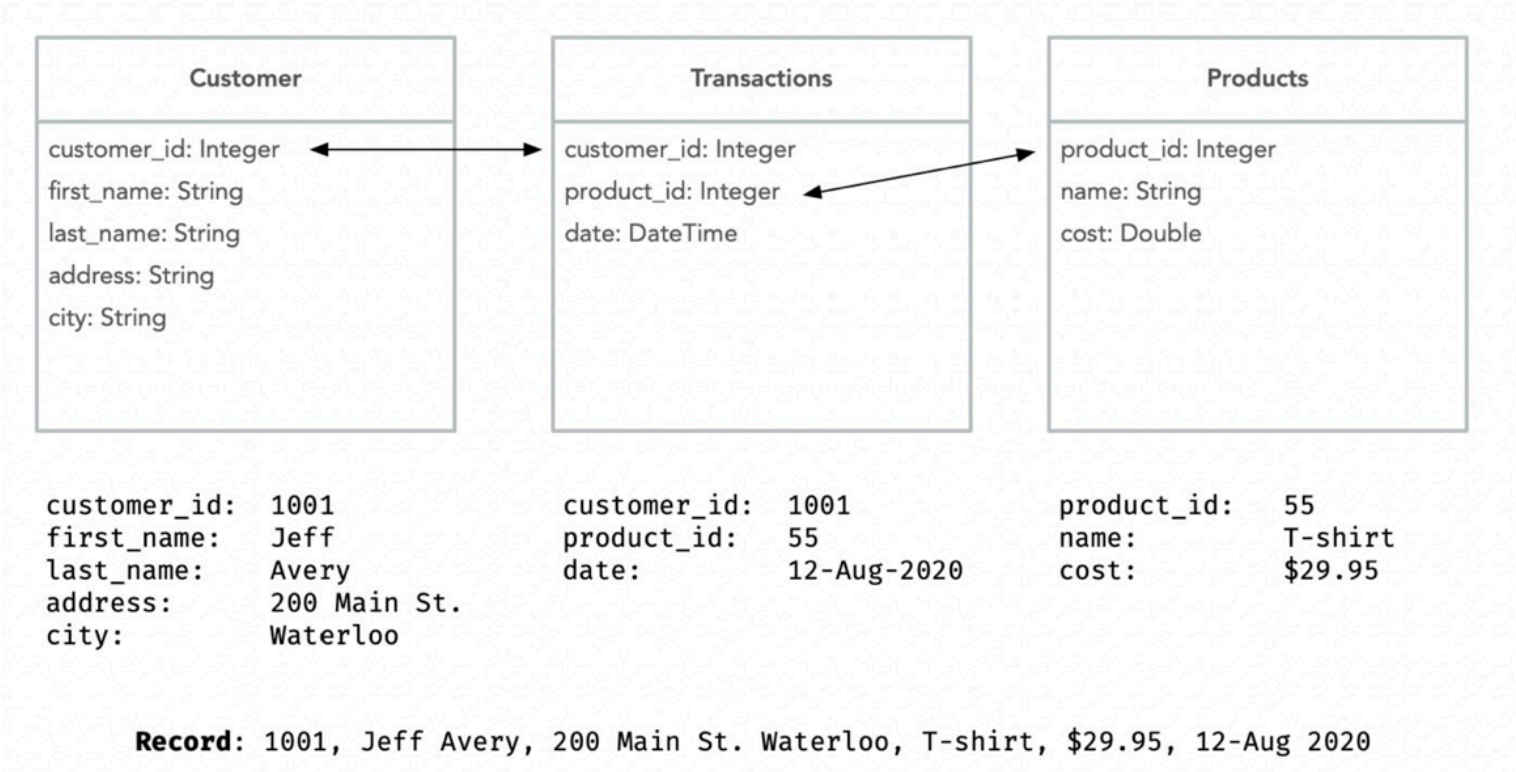

Figure 2: We pull elements from each table to reconstruct a record.

Concepts

Transactions

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

All changes are performed together i.e., they are `atomic`. If there is any error in performing an action, we undo all of these actions [1].

How do you use this?

- "start" a transaction when you perform a series of operations and
- "commit" when you are done.

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.

Concepts

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.

Database Design

Create your database systematically:

1. Decide on the entities to model.
   - Each object/entity is a table.
   - Fields correspond to properties.
   - Let the DB handle key assignment!
   - Optional: Entity-Relationship Diagram (ERD)
2. Normalize the schema to remove inefficiencies.
   - Split up tables to reduce/remove repetition.
   - Consider cardinality i.e., 1:1, 1:M and consider adding intermediate tables.
3. Design and create the tables.
   - UPPERCASE for tables, camelCase for column names.
   - Define data types of each column; pick suitable types and avoid NULL values!

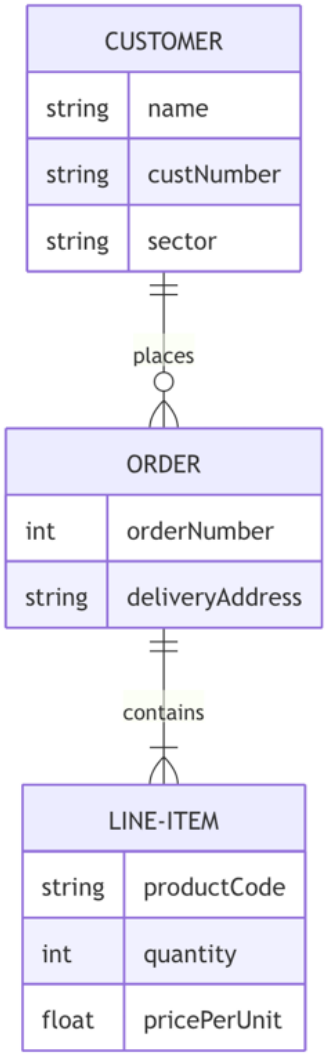

Figure 3: An Entity Relationship Diagram created in Mermaid.

Entity-Relationship Diagrams

An ERD hows entities and 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

- Relationship between rows of different tables.
  - e.g., 1 customer places 0 or more orders.
  - e.g., 1 order contains at least one line item.

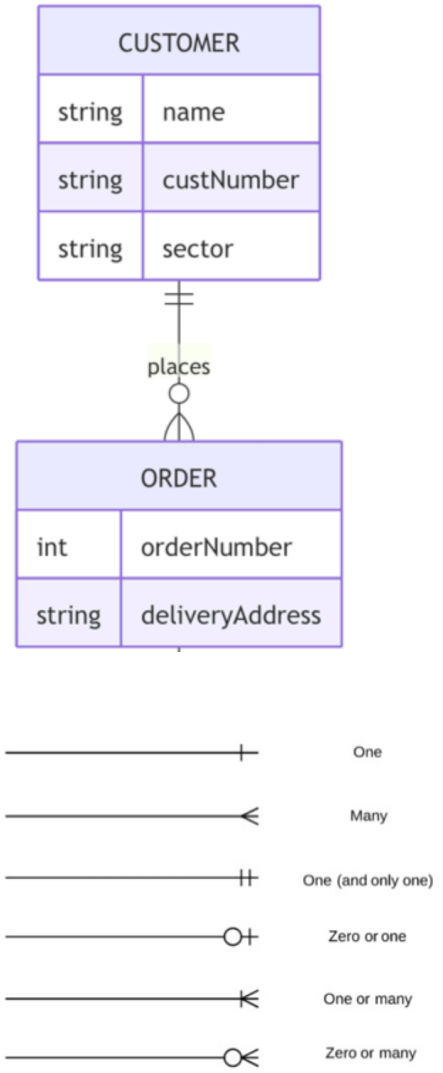

Figure 4: Arrow style indicates the relationship.

Data Types

You may not have the full range of data types that you are accustomed to using in your favorite programming language.

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

SQL

Structured Query Language (SQL) is a Domain-Specific Language (DSL) for describing your queries. It's also an ANSI/ISO standard, and works fairly consistently across relational databases.

You can use SQL to write statements describing the operation to perform and the database performs it for you. You can use it to:

- **Create** new records
- **Retrieve** sets of existing records
- **Update** the fields in one or more records
- **Delete** one or more records

SQL has a very particular syntax [2]:

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

- Operations: `SELECT`, `UPDATE`, `INSERT`, `DELETE`
- Conditions: `<column> <operator> <value>`, for operators `=`, `>`, `<` etc.

SQL

Create: Add A Record

The `INSERT` operation adds a new record 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?
```

SQL

Retrieve: Fetch Existing Records

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

```
SELECT * FROM Customers
-- returns all records

SELECT * FROM Customers WHERE city = "Ottawa"
-- returns records that match 'Ottawa'

SELECT name FROM Customers WHERE custid = 1001
-- returns string "Jeff Avery"
```

> **Tip**
>
> Asterix (*) is a wildcard meaning `all`.

SQL

Update: Modify Existing Records

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

```
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"
-- no `where` clause, so we change all records to `Kitchener`.
```

> **Tip**
>
> To operate on a single row, use a `WHERE` clause that includes the primary key for that row.

SQL

Delete: Remove Records

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

```
DELETE FROM Customer WHERE cust_id = 1001
-- deletes one matching record since cust_id is unique

DELETE FROM Customer
-- deletes everything from this table
```

> **Tip**
>
> To operate on a single row, use a `WHERE` clause that includes the primary key for that row.

Joining Records

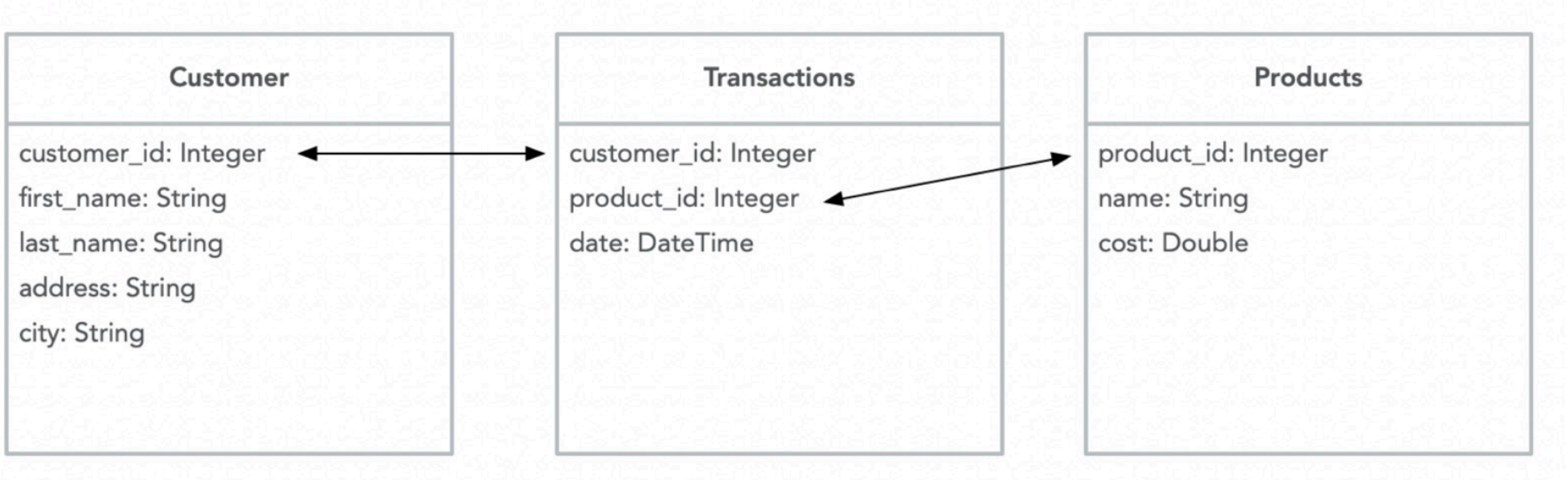

Figure 5: To retrieve a record that spans tables, we need to describe the key relationship between the tables using JOIN syntax.

```
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
```

SQL

Types of Joins

There are different types of joins with different behaviours:

- (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

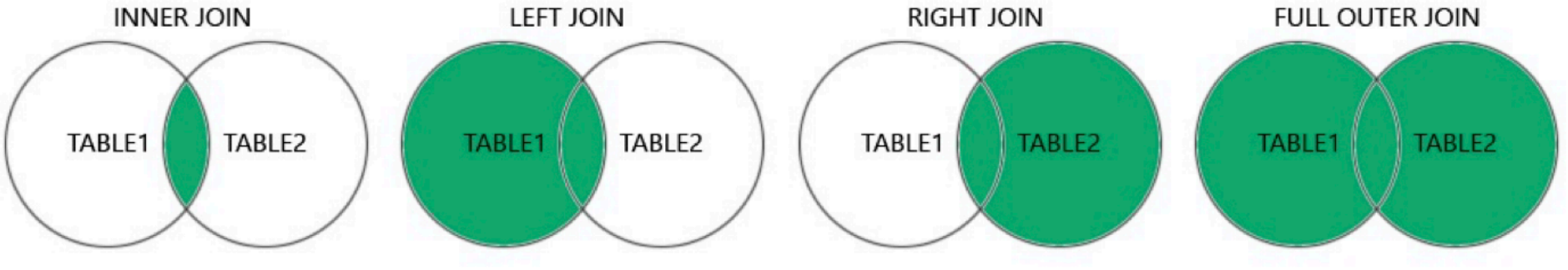

Figure 6: The type of join determines how to handle records when there is no match using a particular relationship.

SQL

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 not show up in the results.

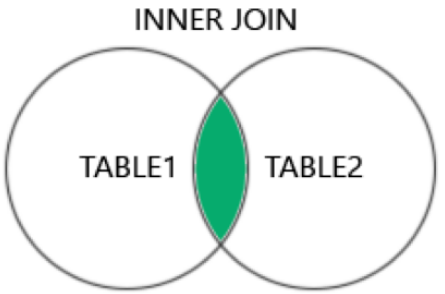

Figure 7: Inner Join

SQL

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.

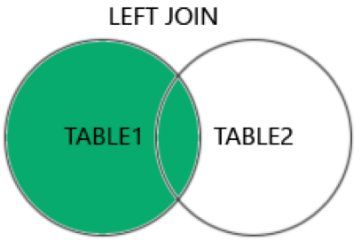

Figure 8: Left (Outer) Join

SQL

RIGHT (OUTER) Join

```
SELECT column_name(s)
FROM table1
RIGHT 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.

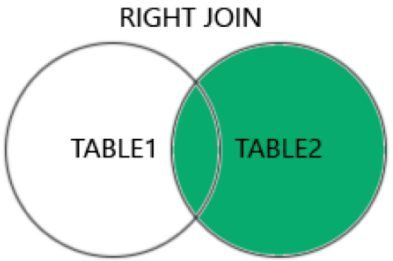

Figure 9: Right (Outer) Join

Example: SQLite

Introduction

> 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.
>
> — https://sqlite.org

SQLite is a relational, file-store database.

- It's preinstalled on macOS, Linux and Android.
- It's a single-user database, by design.
- No authentication; no username/password required.
- It's blisteringly fast and very lightweight.

Most DBMSs are intended to be run in the cloud/network, to facilitate data sharing. SQLite instead is focused on being the best local storage available.

Installation

SQLite can be downloaded and installed on Mac, Windows or Linux. You can interact with it programatically or from the command-line.

To launch the CLI application, run `sqlite` from the command-line. Use `.help` to get a list of commands.

```
$ sqlite3
SQLite version 3.51.0 2025-06-12 13:14:41
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.

sqlite> .help
.archive ...             Manage SQL archives
.auth ON|OFF             Show authorizer callbacks
.backup ?DB? FILE        Backup DB (default "main") to FILE
.bail on|off             Stop after hitting an error.  Default OFF
.cd DIRECTORY            Change the working directory to DIRECTORY
.changes on|off          Show number of rows changed by SQL
...
```

Installation

Useful commands include:

Command	Description
.open `<filename>`	Open a database.
.database	Show all connected databases.
.log <filename>	Write console to log file.
.read <filename>	Read input from file.
.tables	Show a list of tables in the open database.
.schema <table>	Display the SQL create statement for a table.
.fullscheme	Display the SQL to create the entire database.
.quit	Quit and close connections.

Installation

Here's an example of a session against the `chinook.db` database.

```
$ 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

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

Installation

Query:

```
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

Query:

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

ArtistId	Name
1	AC/DC

Installation

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.

Query:

```
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
```

JDBC

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

- Most databases have a JDBC driver available.
  - i.e. Google your database + “JDBC” to locate drivers.
  - locate the “dependency” information.

To add the dependency in your project, modify the `module.yaml` file to include your database driver details:

```
dependencies:
  - org.xerial:sqlite-jdbc:3.50.3.0
```

In your code, use the `Java SQL` package classes to connect and fetch data.

JDBC

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" // format varies by driver
    connection = DriverManager.getConnection(url)
    println("Connection to SQLite has been established.")
  } catch (e: SQLException) {
    println(e.message)
  }

  return connection
}
```

JDBC

Running a query:

```
fun query(connection: Connection?) {
  try {
    if (connection ≠ null) {
      val st = "select albumId, title from albums where albumid < 5"
      val query = connection.createStatement()
      val results = query.executeQuery(st)

      while (results.next()) {
        val albumId = results.getInt("albumid")
        val title = results.getString("title")
        println(albumId.toString() + "\t" + title)
      }
    } catch (ex: SQLException) {
      println(ex.message)
    }
}
```

```
1 For Those About To Rock We Salute You
2 Balls to the Wall
3 Restless and Wild
4 Let There Be Rock
```

JDBC

Why NOT JDBC?

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.

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.

Room

Google created Room in 2017, as an abstraction layer over SQLite.
Room + SQLite is Google's recommended solution for Android.

Room can be installed like any other KMP library. It's multiplatform, so you can use it on Android, Desktop and iOS.

Install the Room dependencies in your shared `module.yaml` file.

```
dependencies:
  - androidx.sqlite:sqlite-bundled:2.7.0
  - androidx.room:room-runtime:2.8.4
  - androidx.room:room-compiler:2.8.4
```

Room

Room provides an abstraction layer that sits above JDBC calls.

Like many database abstraction layers, it's main benefit is to reduce complexity and runtime errors that you risk with low-level libraries.

- You gain compile time verification of SQL queries.
- Missing tables and other structural problems are identified at compile time.
- Types agreement between entities and methods that use them.

There are three core classes in Room. Your application will need to implement each of these [3]:

1. The `Database` class that holds the database and serves as the main access point for the underlying connection to your app's persisted data.
2. `Entities` that represent tables in your application's database.
3. `Data access objects (DAOs)` that provide methods that your application can use to query, update, insert, and delete data in the database.

Room

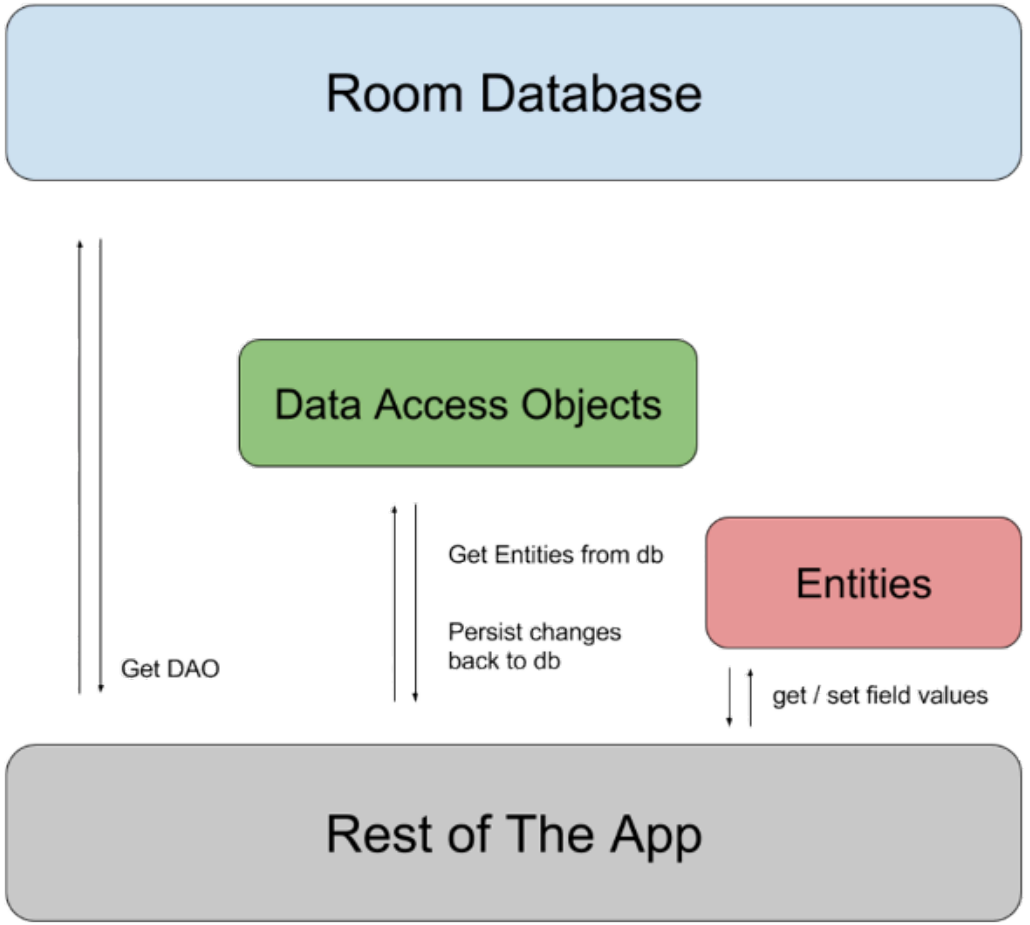

Figure 10: Room library architecture, courtesy of [developer.android.com]

Examples

@Entity

This is a Domain Object that reflects rows in a table (i.e., table structure). It's 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
)
```

Examples

@DAO

This is a Data Access Object that represents the underlying data. It exposes data methods that work with your entities, and maps your operations to remote database methods e.g., delete, select.

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

  @Delete
  suspend fun deleteContact(contact: Contact)

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

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

Examples

@Database

This represents the main access point to the database. It implicitly instanatiated and manages the Dao for you (suggested by this code; you don't need to actually override the `dao` method here).

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

Examples

Example: Android

The `Taskit` application implements Room for local storage. Here's part of the implementation code to illustrate how this might work.

Our entity class models a table named `task_table` in the DB.

`TaskEntity.kt`

```
@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
)
```

Examples

The DAO class handles interactions with the database

`Taskdao.kt`

```
@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)
}
```

Examples

The application class constructs the screen using this data.

Application.kt

```
@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
        }
      )
    }
  }
```

Examples

Finally, we bind everything together in the MainActivity.

MainActivity.kt

```
/*
The Application screen only accesses data through the ViewModel.
*/

val database: TaskDb = getRoomDatabase(this)
val model = Model(database.taskDao())
val viewModel = TaskViewModel(model)

setContent {
  Application(viewModel)
}
```

Examples

When to use Room?

You should consider using Room + SQLite for local operations.

- Works best when your application has exclusive access to a database.
- You will need SQLite database drivers for each platform that you intend to target.

When NOT to use Room?

Room is not the best choice when you need non-exclusive (shared) access.

- You would need to build out a service to manage shared access to the database; there are better ways to accomplish this.

It’s also SQLite-only. If you need a different database, you will have to consider a different access option.

Example: Supabase

Introduction

Rosenberg & Mateos [4] define cloud computing as:

"[A] model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services)".

You can certainly install and host a database locally, but there are strong reasons to push your database management system into the cloud.

These include:

- **Availability**: your data becomes available to any authorized users.
- **Flexibility**: you can deploy new hardware with a button-click.
- **Scalability**: you can grow your hardware with demand e.g., scale up during peak season, scale down when demand shrinks.
- **Security**: you can outsource security management, to some degree, to a third party. This is often safer than managing it in-house.

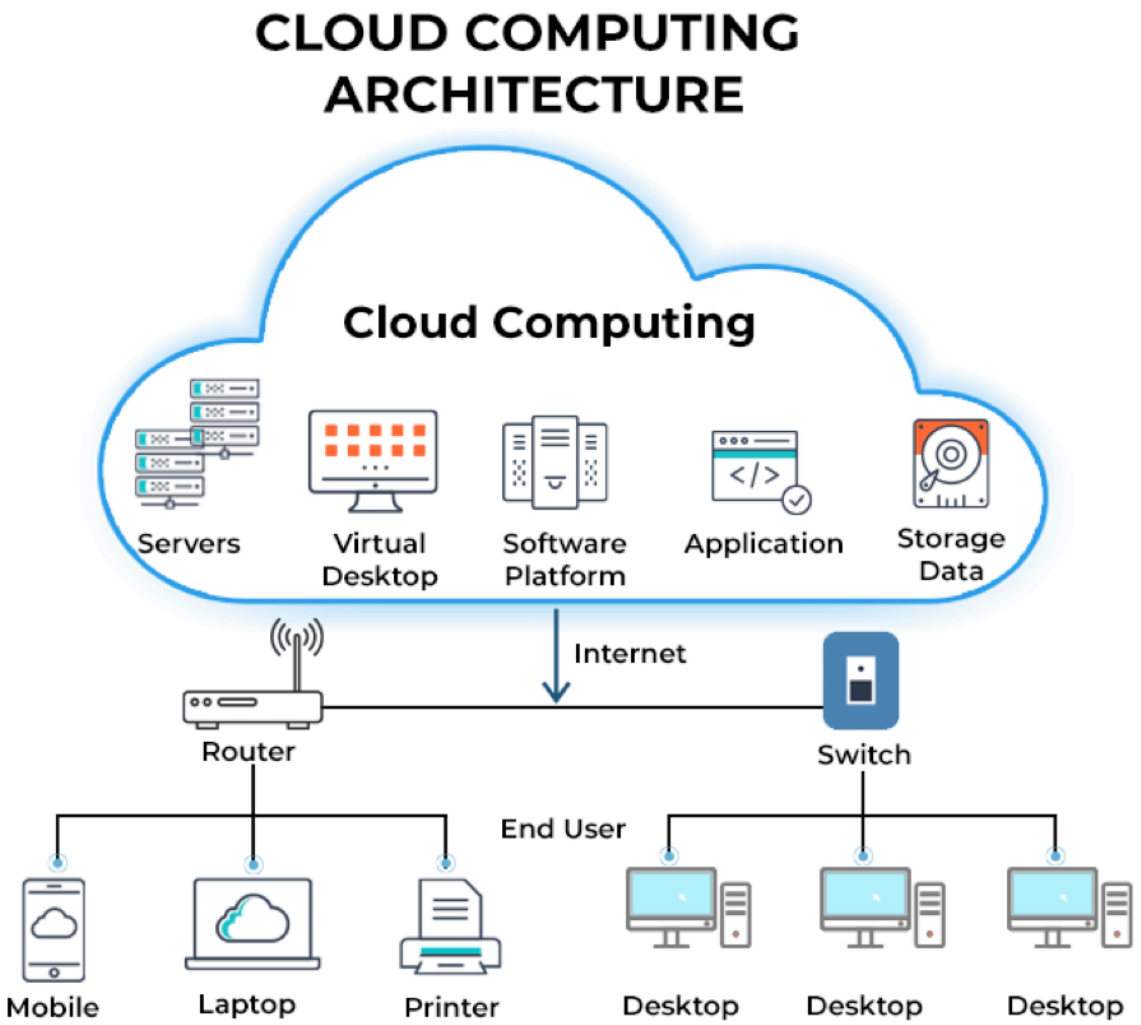

Figure 11: Cloud products like Microssoft Azure, or Google Cloud offer always-on cloud services to meet practically any need. Diagram courtesy of Spiceworks.

Introduction

What do we need for our applications?

An online, shared, database

- A relational database where we can store/retrieve application data.
- It should also have the ability to manage binary data e.g., images, video.

Managed access control

- Authentication: ability to verifying the identity of a person or system (e.g., username/password).
- Authorization: managing and restricting which resources a user can access, and what capabilities are available e.g., read but not write table X.

Notifications

- A mechanism to receive updates from the platform when data changes.
- Necessary when multiple users can access and modify data simultaneously.

Introduction

You have a large and growing number of options to address this requirement.

Document databases:
- MongoDB
- Google Firebase

Relational databases:
- Amazon AWS
- Microsoft Azure
- Google Cloud Platform
- Neon
- Supabase

We recommend Supabase because
- it's "just" a database, so we're not left managing services that we don't need.
- the backing database is Postgres, which has become a standard (of sorts).
- they have a free/no-cost tier for hobbyists and students.
- they have a fully supported Kotlin KMP access library!

Installation

You can setup your database and authorization directly on the Supabase website. Make sure to use their “free” tier.

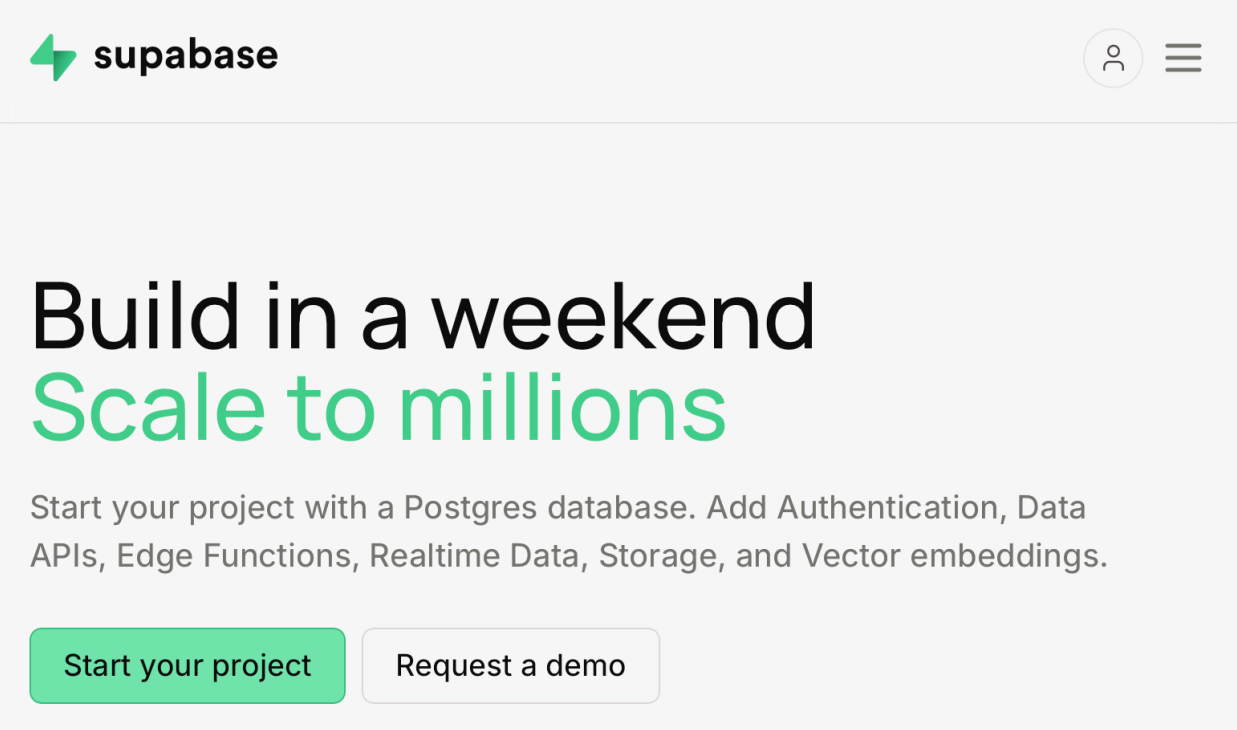

Figure 12: Supabase Home Page

Step 1. Create a new project

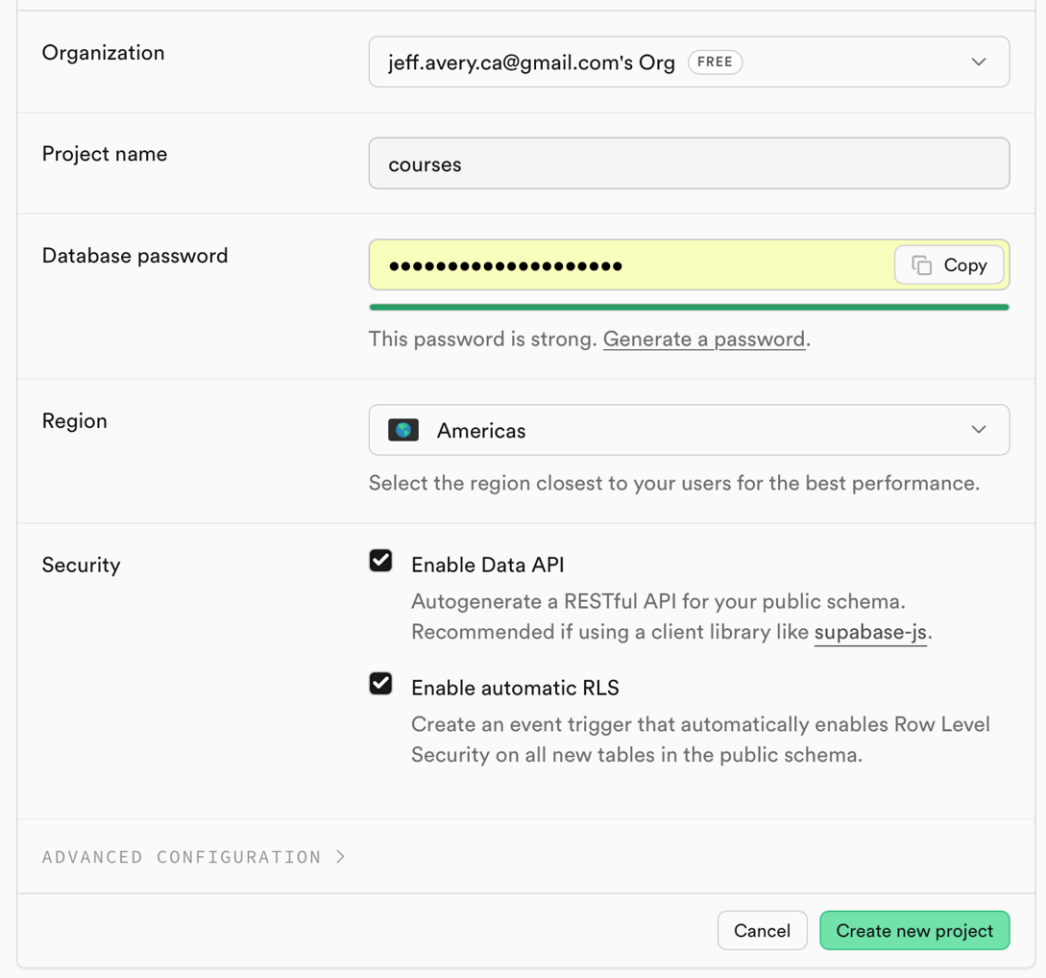

Figure 13: Nagivate to www.supabase.com and choose ‘Start your project’. A wizard will walk you through the steps.

Installation

Step 2. Configure access

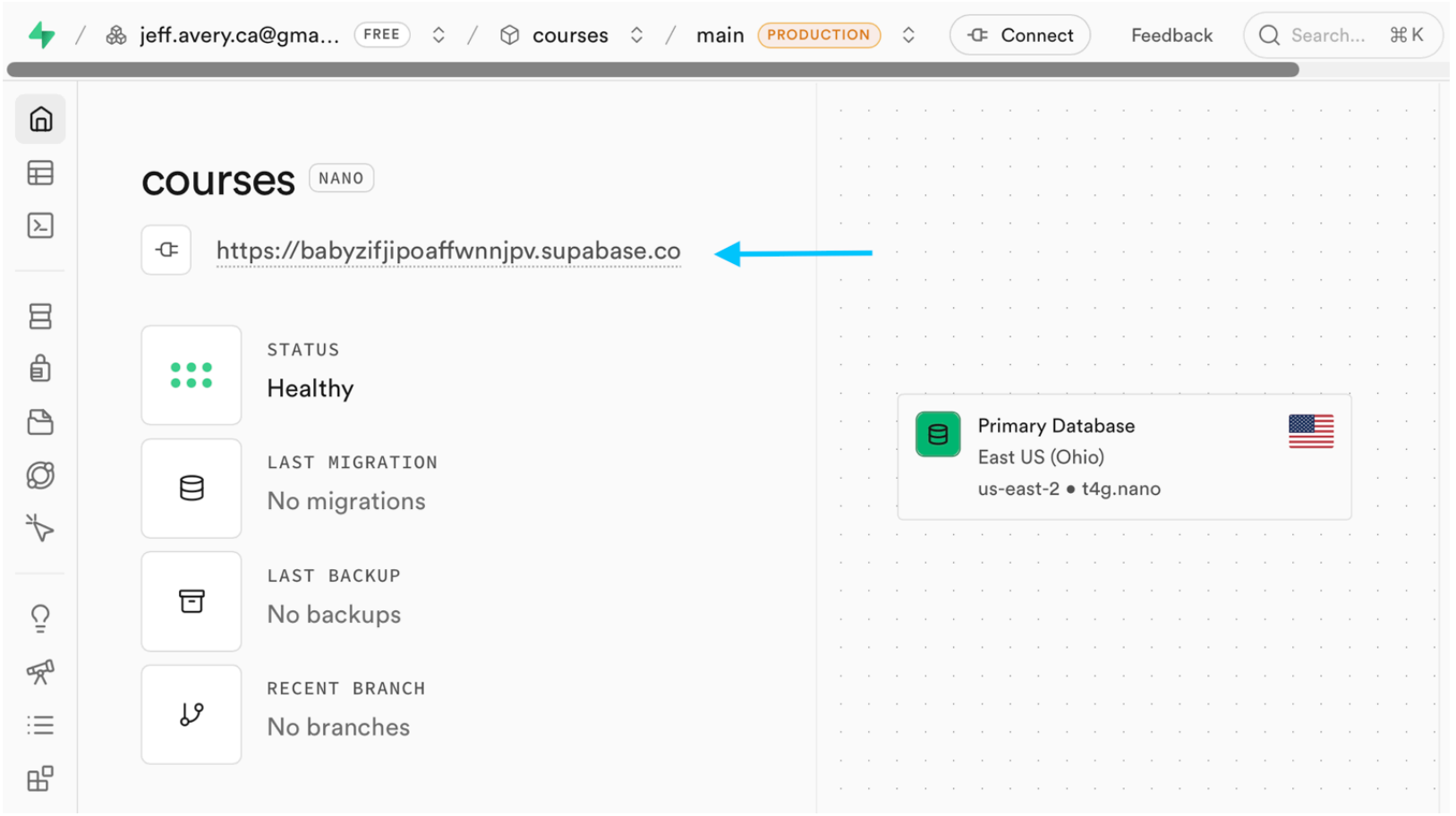

Figure 14: This URL is the endpoint to your database. Click on it to configure access.

Installation

Step 3: Copy your URL & Key

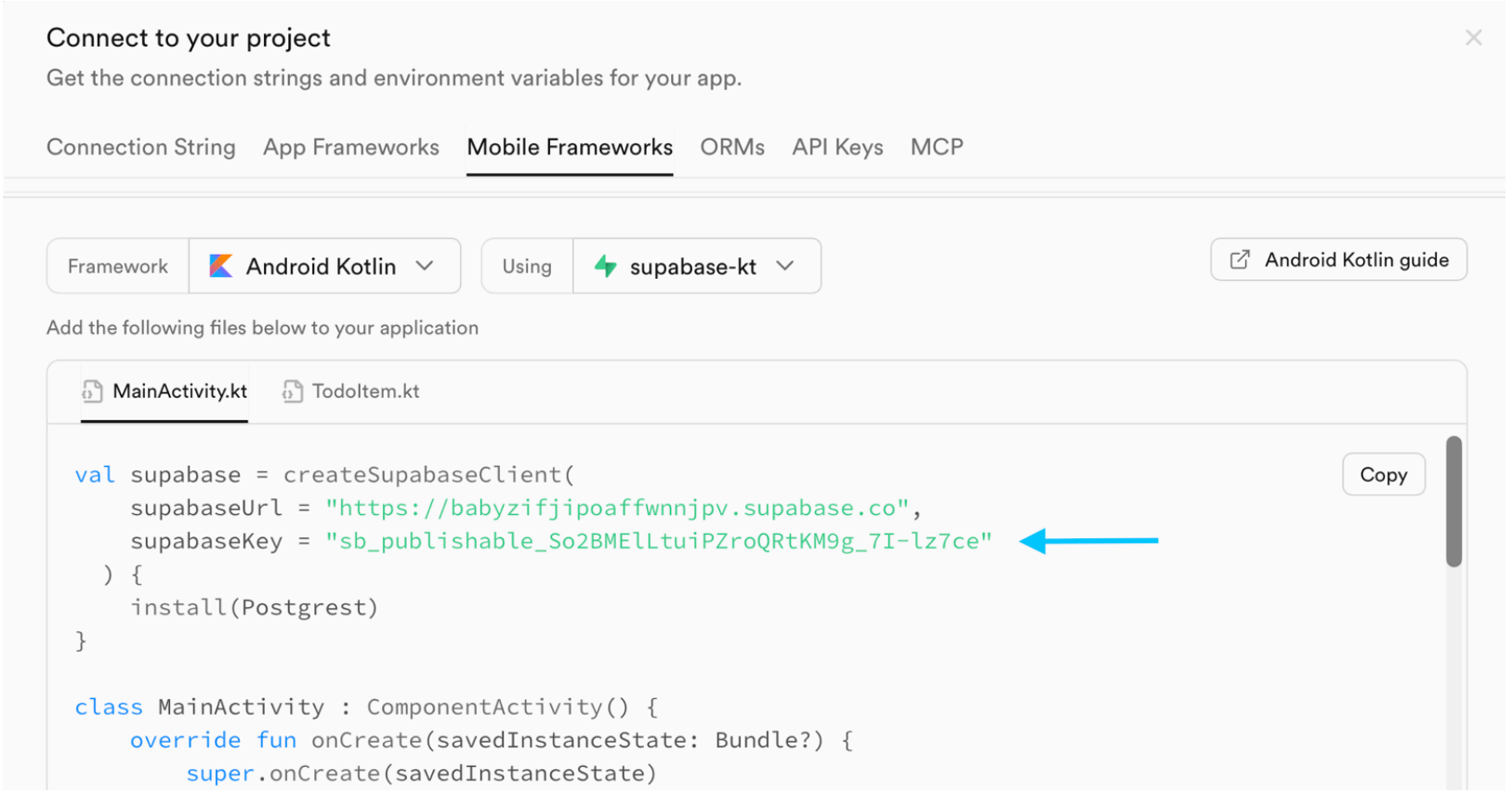

Figure 15: The Mobile Frameworks tab provides sample code for accessing your DB!

Installation

Setp 4: Create your schema

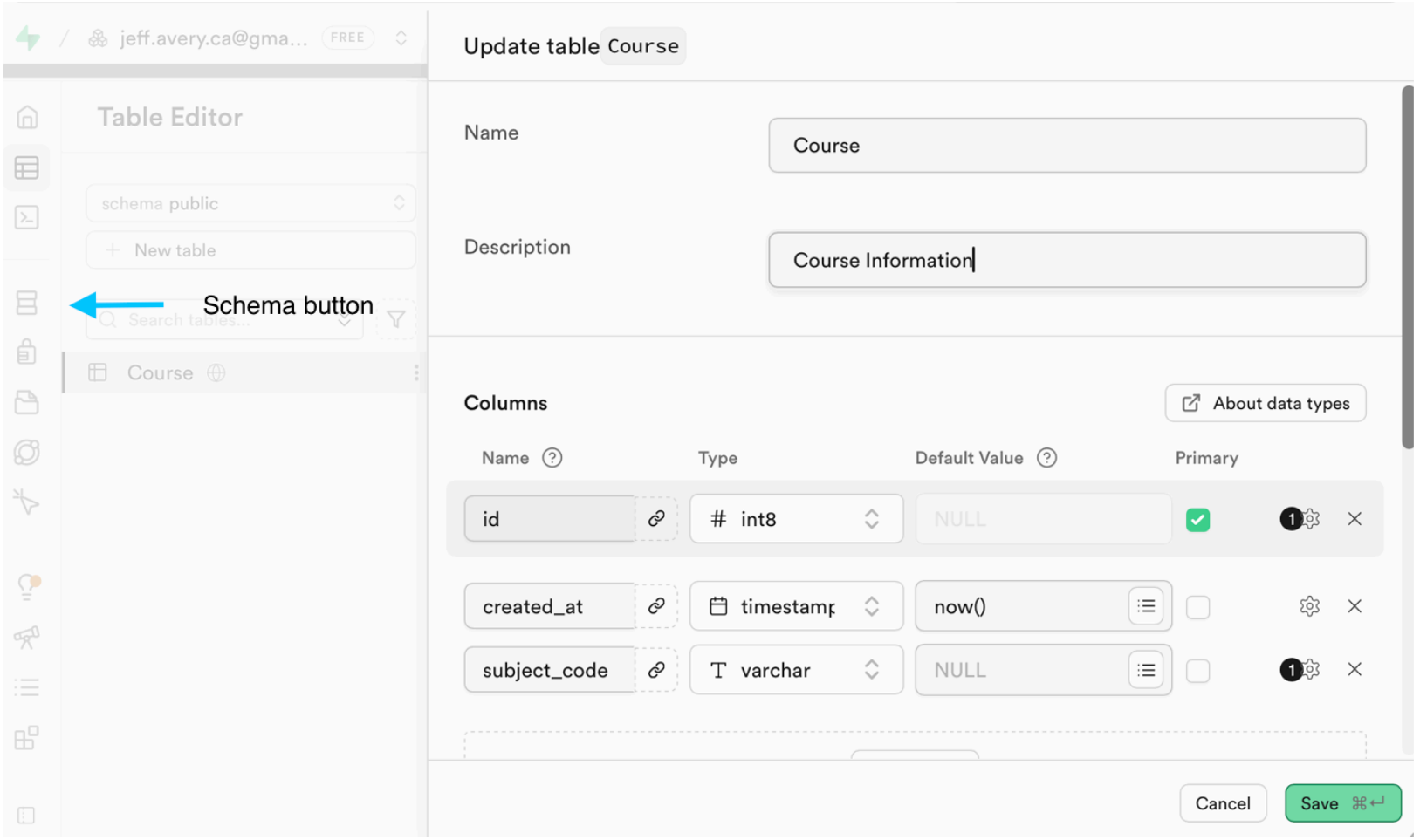

Figure 16: Create your table structure from the Schema section.

Installation

Step 5: Setup access

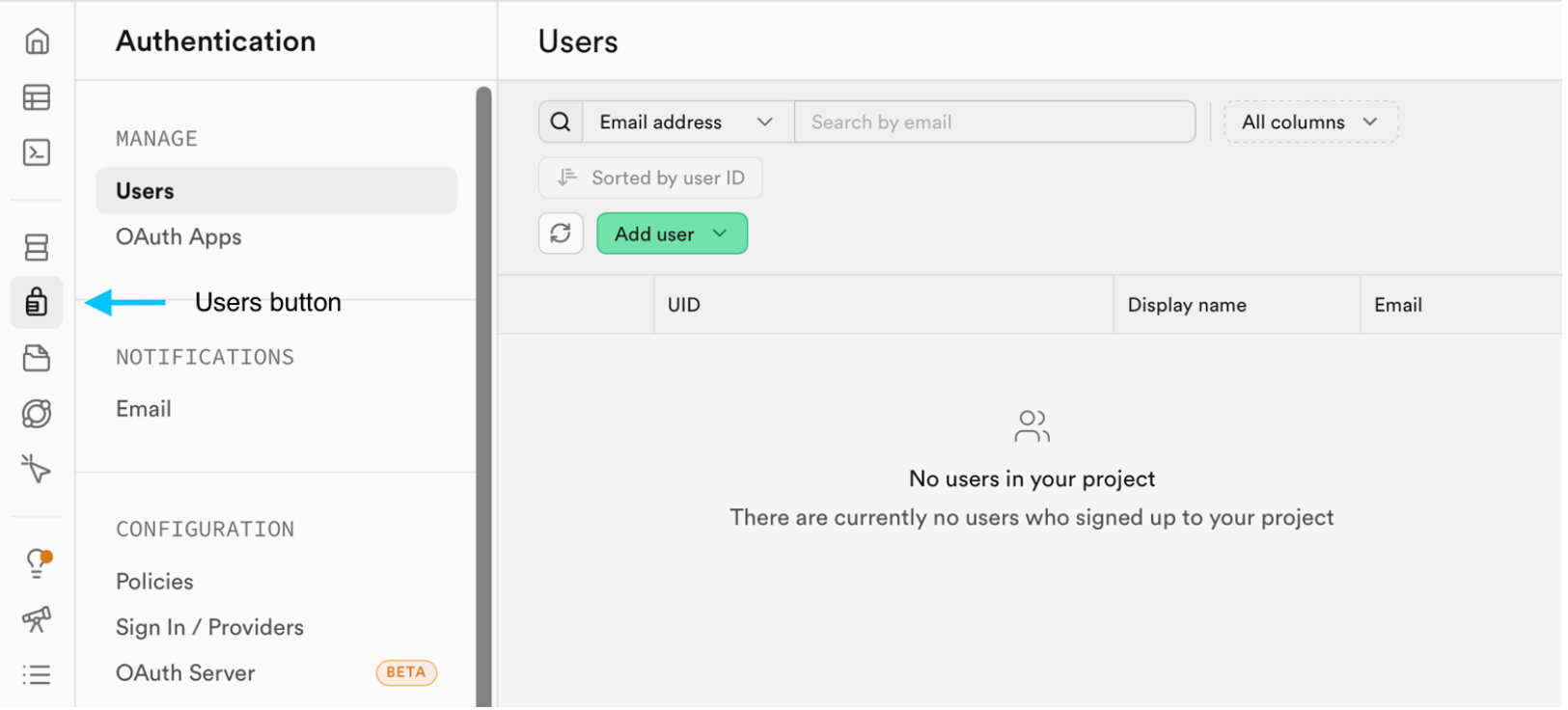

Figure 17: Define users and what permissions they have for your database.

Kotlin SDK

Supabase has a Kotlin SDK which supports connections from a number of supported platforms, including Android, iOS and Desktop.

> You can use supabase-kt to interact with your Postgres database, listen to database changes, invoke Deno Edge Functions, build login and user management functionality, and manage large files

[5]

To install the libraries, see the Installation section of the SDK.

- Install the listed libraries by adding them to your `module.yaml` file for your shared module (for KMP projects). e.g.,

```
dependencies:
  - io.github.jan-tennert.supabase:bom:3.6.0
  - io.github.jan-tennert.supabase:postgrest-kt
  - io.github.jan-tennert.supabase:auth-kt
  - io.github.jan-tennert.supabase:realtime-kt
```

Examples

These code samples use the Kotlin SDK and the `url` and `key` provided in your project. See the [5] for many more examples.

`mm-desktop`: connecting to a database [5].

```
class CloudStorage : IStorage {
  var supabase: SupabaseClient? = null
  var auth: Auth? = null
  init {
    supabase = createSupabaseClient(
      supabaseUrl = "https://bdifjfbvufwqdqfsdwse.supabase.co",
      supabaseKey = "sb_publishable_HaCdYvgF1YgfUWSqwIrTwg_EZ8VGdLb"
    ) {
        install(Postgrest)
        install(Auth)
      }
    auth = supabase?.auth
  }
}
```

Examples

Managing Credentials

`mm-desktop`: handling user credentials [5].

```
suspend fun createUser(email: String, password: String) {
  supabase?.auth?.signUpWith(Email) {
    this.email = email
    this.password = password
  }
}

suspend fun loginUser(email: String, password: String) {
  supabase?.auth?.signInWith(Email) {
    this.email = email
    this.password = password
  }
}

suspend fun logoutUser() {
  supabase?.auth?.signOut()
}
```

Examples

Running a Query

`mm-desktop`: running a query using the Kotlin Client library [5].

```
override suspend fun read(id: Int): Task? {
  return
    supabase?.from("tasks")
      .select(columns = Columns.list("id", "priority", "title")) {
        filter { Task::id eq id }
      }?.decodeSingle<Task>()
}
```

Bibliography

[1] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*. McGraw Hill, 2021.

[2] Unknown, “SQL Keywords Reference.” [Online]. Available: https://www.w3schools.com/sql/sql_ref_keywords.asp

[3] Google Inc., “Save data in a local database using Room.” [Online]. Available: https://developer.android.com/training/data-storage/room

[4] J. Rosenberg and A. Mateos, *The Cloud at Your Service - The When, How, and Why of Enterprise Cloud Computing*. Manning Publications, 2010.

[5] Supabase Inc., “Kotlin Client Library.” [Online]. Available: https://supabase.com/docs/reference/kotlin/introduction