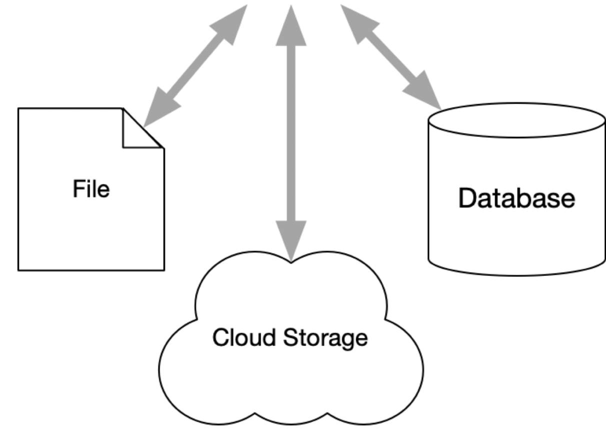
Relational & NoSQL Databases

CS 346: Application Development

What is a database?

A database is a **system for storing data** (often represented as records).

- Advantages over just storing data in files?
  - It's designed for working with **large numbers of records**.
  - Databases support **efficient and complex operations** on your records.
  - Facilitates **data sharing** across large numbers of concurrent users/systems.
- There are many kinds of databases:
  - Relational, Graph Databases, Document Databases…
- We'll focus on Relational and NoSQL/Document.



Local vs. Remote Data

Databases can be stored **locally** (the same system as your application), or **remotely** (over a network to a different system).

Local

- Often done when you don't need to share data between applications or users.
- e.g., I have an application that stores my clipboard history. For practical reasons (security, performance) I don't need this shared with anyone else. The data is stored in a local database.

Remote

- Done when you need to share data with other users or applications.
- e.g., messaging systems like Slack need to share data between users.

We'll *mostly* ignore this distinction for now (and revisit when discussing services/cloud). Just remember that you might use a local database for reasons other than data sharing.

Relational Databases

Old-school, SQL databases. Still incredibly useful.

Introduction

Relational databases are based on the relational model proposed by Dr. E.F. Codd.

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

- Relational databases are exceptional at storing and processing structured data. They are hugely popular e.g., Oracle, SQL Server, PostgreSQL, MySQL, Supabase.

What are the benefits of relational databases?

1. Relational databases allow for very efficient data storage i.e. little redundancy.

2. Relational databases support operations on **sets** of records, which mirrors how we work. e.g.
    - *Fetch a list of all purchases greater than $100.*
    - *Display customer_name for all customers that live in "Ottawa".*

3. They are *declarative*, meaning that the DB performs named operations without you needing to understand how they are implemented.

Table

A **table** is the foundational concept. It collects related **fields** into **records.**

Roughly analogous to a class, with each row a record/instance.

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

Field (Column) names

Records (Rows)

Data -> Tables

Class

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

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

CSV File

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

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

Database Table

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

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

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.

- `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

Foreign Keys

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

A primary key in 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

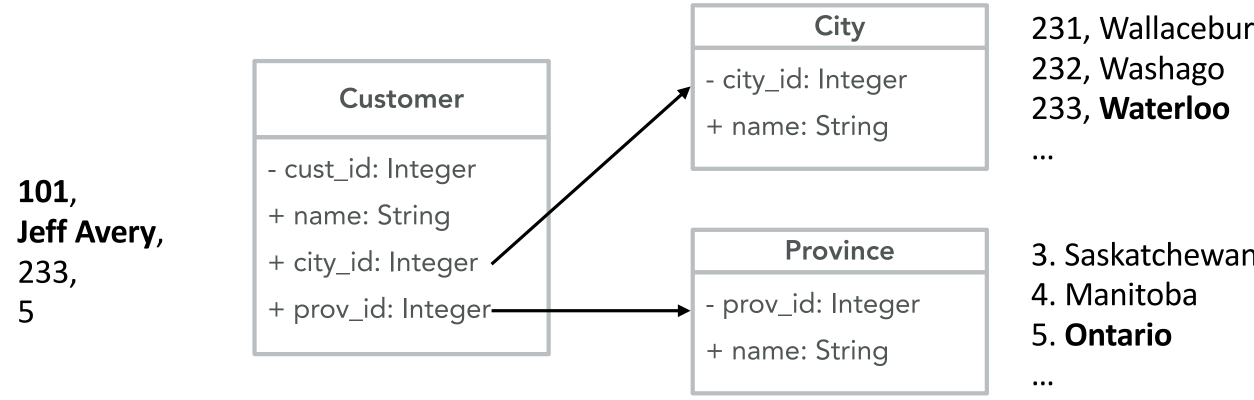
tx_id	cust_id	Item	Amount
43222	1002	Chemistry set	125.00
54187	1003	Duct tape	5.99

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

Joins

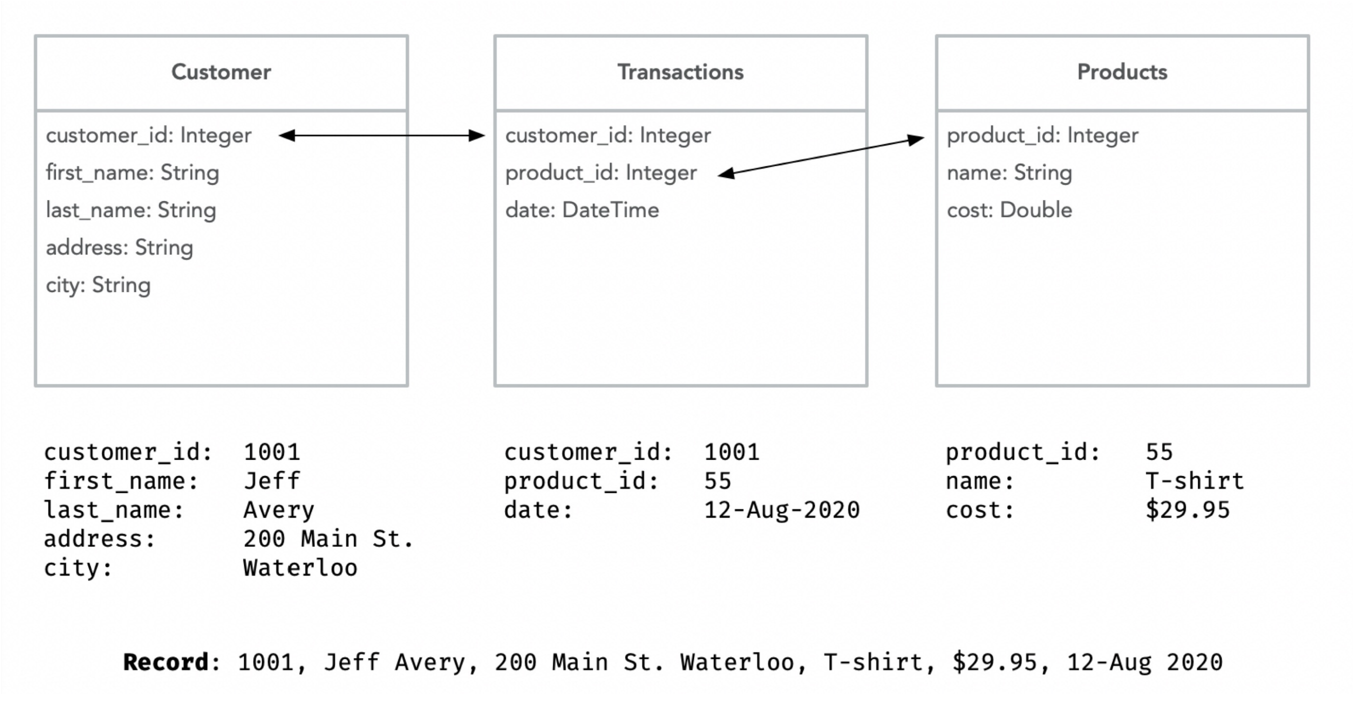
A well-designed database avoids data duplication. This means that we want to split unique entities into their own tables.

- e.g., a Customer record like `“101, Jeff Avery, Waterloo, ON”` would be split apart:



Joins

A `join` describes how to relate data across tables using keys. We split data for efficiency but joins lets us reassemble records when we need that information again.



ACID

In computer science, **ACID** (atomicity, consistency, isolation, durability) is a set of properties intended to guarantee data validity despite errors, power failures, and other mishaps.

- **Atomicity** prevents updates to the database from occurring only partially.
- **Consistency** guarantees that transaction can move database from one valid state to the next. This ensures these all adhere to all defined database rules. Also preventing corruption by illegal transaction.
- **Isolation** determines how a particular action is shown to other concurrent system users.
- **Durability** is the property that guarantees that transactions that have been committed will survive permanently.

Transactions

Transactions ensure ACID properties.

- 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 any of those actions, we undo *all* of these actions. We commit everything, or rollback everything.
- How do you use this? You “start” a transaction when you perform an operation and “commit” when you are done.
  - This is often done implicitly – details will vary by database/library (so check!)
- This is how we can handle two or more users are updating the same data at the same time and ensure consistency of data.

SQL – manipulating Data

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

You can also choose to use libraries which perform actions without SQL. This is still a great model for thinking about what operations to perform.

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
```

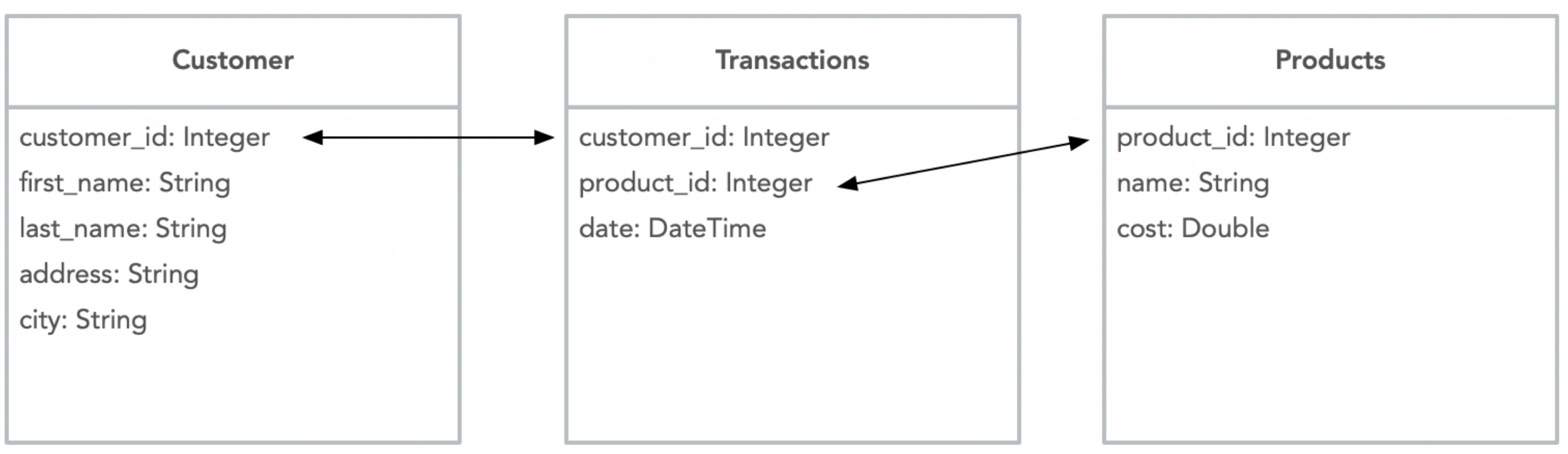
Filtering with a "Where" clause

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

```
UPDATE Customer SET city = "Kitchener" WHERE cust_id = 1001
—> updates one record since cust_id is unique for each row
```

WHERE also allows us to define relations between tables.

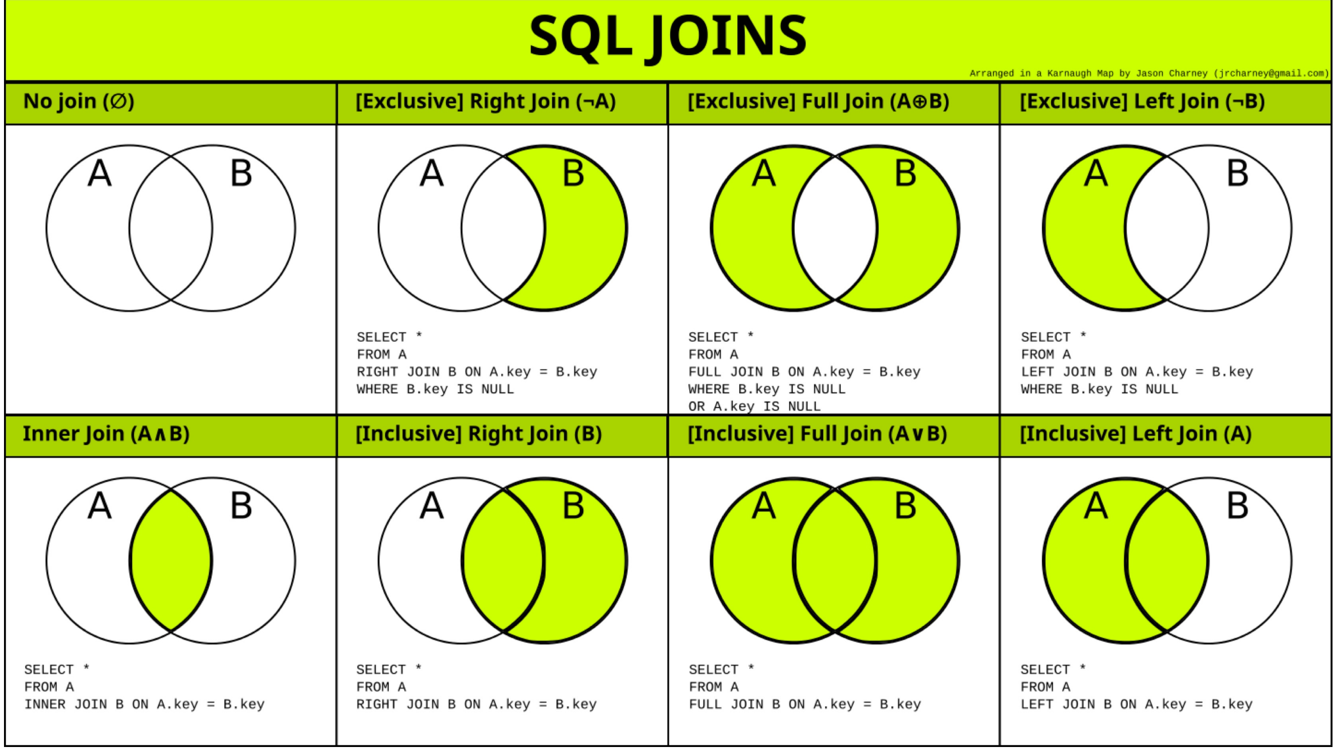
This means that we can start to run more complex queries across multiple tables.




```
$ 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
```

Types of Joins (Syntax)



Accessing SQL Databases in Kotlin

Kotlin leverages the Java JDBC API (”Java DataBase Connectivity”). This provides a standard mechanism for connecting to databases, issuing queries, and managing the results. To create a database project in IntelliJ:

1. Create a Gradle/Kotlin project.

2. Modify the build.gradle.kts to include a dependency on a suitable JDBC for your specific database (e.g. MySQL, PostgreSQL, SQLite).

```
implementation(“org.xerial:sqlite-jdbc:3.39.3.0”) // example of SQLite
```

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

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

Creating a connection

This example uses a sample database from the SQLite tutorial.
We first create a connection to the database. The URL designates both the type of database, and location of the local database file.

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

Running a Query

```
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.
```

DB Abstraction

JDBC is a useful mechanism for connecting to remote databases, but making raw SQL calls is error-prone: there's no type checking, or other safety mechanisms in-place. It also requires us to explicitly convert between string data and class objects that are holding our data.

- There are several libraries that abstract the complexities JDBC and provide a cleaner database interface. Popular ones include:
  - Exposed is a JetBrains Kotlin library for working with JDBC databases. It works great with desktop applications (local or remote).
  - Room is an Android library for working with SQLite databases (usually locally).

Exposed

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

- typesafe SQL wrapping DSL, or
- lightweight Data Access Objects (DAO).

It works through JDBC and supports most popular databases (incl. SQLite, H2, Oracle, Postgres…)

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

Exposed Example (1/2)

```
object CoursesTable : Table() {
    val courseID: Column<String> = varchar("course_id", length = 10)
    val term: Column<Int> = integer("term")
    val termName: Column<String> = varchar("term_name", length = 50)
    val subject: Column<String> = varchar("subject", length = 10)
    val catalogNumber: Column<String> = varchar("catalog_number", length = 10)
    val title: Column<String> = varchar("title", length = 100)
    val description: Column<String> = varchar("description", length = 1024)
    val requirements: Column<String> = varchar("requirements", length = 1024)
    override val primaryKey = PrimaryKey(courseID, name = "pk_course_id")
}

private fun createTables() {
    transaction {
        SchemaUtils.create(CoursesTable, SectionsTable)
    }
}
```

https://git.uwaterloo.ca/j2avery/courses

Exposed Example (2/2)

```
fun addCourse(course: Course) {
    transaction {
        CoursesTable.insert {
            it[courseID] = course.courseID
            it[term] = course.term
            it[termName] = course.termName
            it[subject] = course.subject
            it[catalogNumber] = course.catalogNumber
            it[title] = course.title
            it[description] = course.description
            it[requirements] = course.requirements ?: ""
        }
    }
}
```

No-SQL Databases

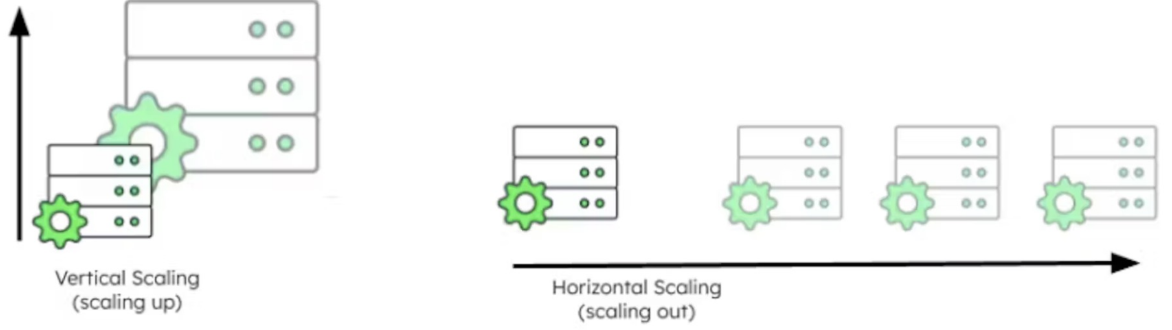
The highly scalable alternative.

Introduction

No-SQL is a very broad category, which can either mean "Not Only SQL" or "No SQL". These databases are designed to handle data that doesn't fit well into the traditional relational model (typically non-structured data, where records may have different structures)

- Types of No-SQL databases include:
  - Document databases (e.g. MongoDB, Google Cloud Firestore)
  - Key-value stores (e.g. Redis)
  - Graph databases (e.g. Neo4j)
  - Time-series databases (e.g. InfluxDB)

NoSQL databases are popular in large-data scenario, where you may need to process extremely large amounts of data and/or your data will grow significantly over time.



SQL databases are often scaled-up as data needs increase (more memory, processing on the same system). NoSQL databases can be scaled out (distributed across more systems).

https://www.mongodb.com

Document Databases

A **document database** is a type of NoSQL database that can be used to store and query data as ***JSON-like documents***.

Why is this a useful paradigm?

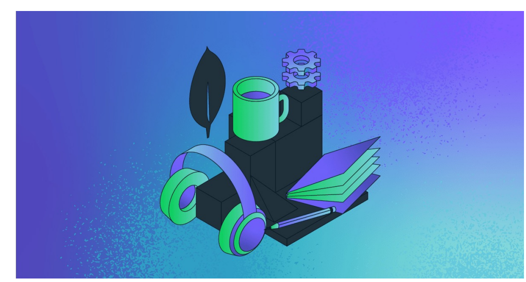
- Easy to develop! Objects (code) map to JSON (data format), which you can then easily store directly in the database.
- Flexible schema. You can add fields to a document at any time.
- Scales out very well if needed for large, distributed systems.

We'll use MongoDB as an example. See MongoDB Community Edition.

Firestore is also popular for Android development. See Firebase Documentation.

Mongo DB

- Can be installed locally, but more often used in the cloud (where they host it for you).
- Exclusively a document database
  - NO table structure
  - JSON documents instead, with flexible structure.
- How do you query it?
  - No SQL!
  - Mongo has a proprietary API that you can use, with a custom driver.
  - Same operations you would expect (CRUD).



See Getting Started with the MongoDB Kotlin Driver.

Adding Mongo Dependency

Add the drivers to your `build.gradle.kts` file.

```
dependencies {
    // Kotlin coroutine dependency
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4")

    // MongoDB Kotlin driver dependency
    implementation("org.mongodb:mongodb-driver-kotlin-coroutine:4.10.1")
}
```

Working with Data

To use the hosted solution, you need to:

1. Setup an account and an online cluster that you can use (free is fine!)
2. Use the connection details to connect to the database from your code.

```
val connectionString = "mongodb+srv://<username>:<enter your
password>@cluster0.sq3aiau.mongodb.net/?retryWrites=true&w=majority"
val client = MongoClient.create(connectionString = connectString)
```

See Getting Started with the Kotlin Driver for setup.

```kotlin
suspend fun setupConnection(
    databaseName: String = "sample_restaurants",
    connectionEnvVariable: String = "MONGODB_URI"
): MongoDatabase? {
        val connectString = "mongodb+srv://<usename>:<password>@cluster0.sq3aiau.mongod
    }

    val client = MongoClient.create(connectionString = connectString)
    val database = client.getDatabase(databaseName = databaseName)
    return try {
        // Send a ping to confirm a successful connection
        val command = Document("ping", BsonInt64(1))
        database.runCommand(command)
        println("Pinged your deployment. You successfully connected to MongoDB!")
        database
    } catch (me: MongoException) {
        System.err.println(me)
        null
    }
}
```

https://www.mongodb.com/developer/products/mongodb/getting-started-kotlin-driver/

Choosing a Database System

What to choose? What things to consider.

What should you choose?

Your choice of database depends on a few factors:

- SQL or No-SQL – choose based on your data needs
  - SQL is much better for structured data (i.e., you can query it, sort it).
- Local or hosted (cloud) – do you need to share data?
  - If you pick local, you will need to distribute it with your application.
  - If you pick cloud, you need to be concerned with security and connectivity.

SQL Databases	Local	Cloud
Android	SQLite	Supabase
Desktop	SQLite	Supabase

NoSQL Databases	Local	Cloud
Android	(ScyllaDB, Cassandra)	Firebase
Desktop	(ScyllaDB, Cassandra)	MongoDB