Data Persistence

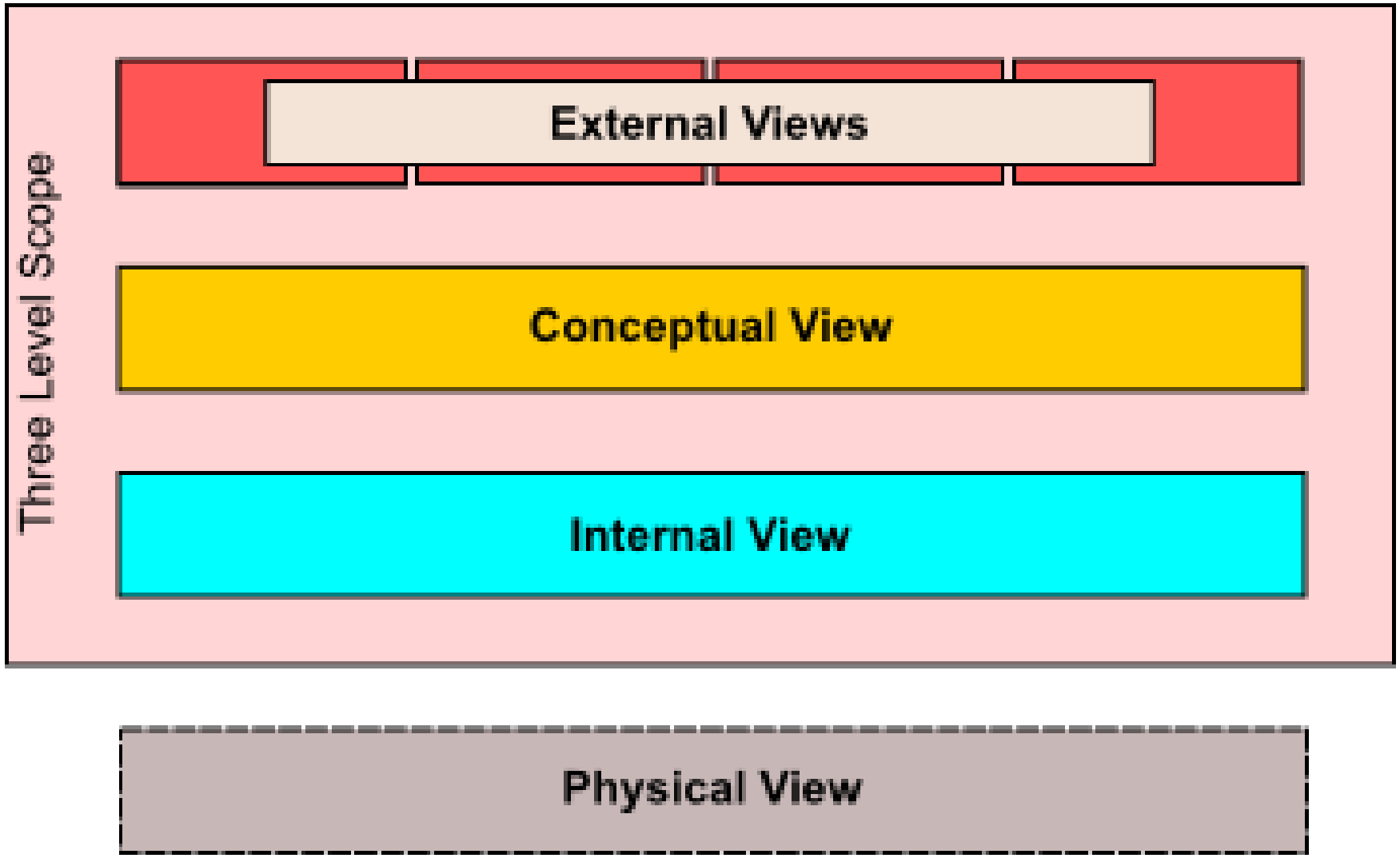
Ways of Thinking about Data

- Data exists in three layers
- The internal model of the data a group of user have
  - Their view of the data based on what they need for their context
  - These are often the informal business objects
- The conceptual model
  - Often called the logical model
  - A rigorously defined description of the data
  - Often represented in some schema
- The implementation model
  - How the data is organized for specific uses
  - Relational model, dimensional model, etc

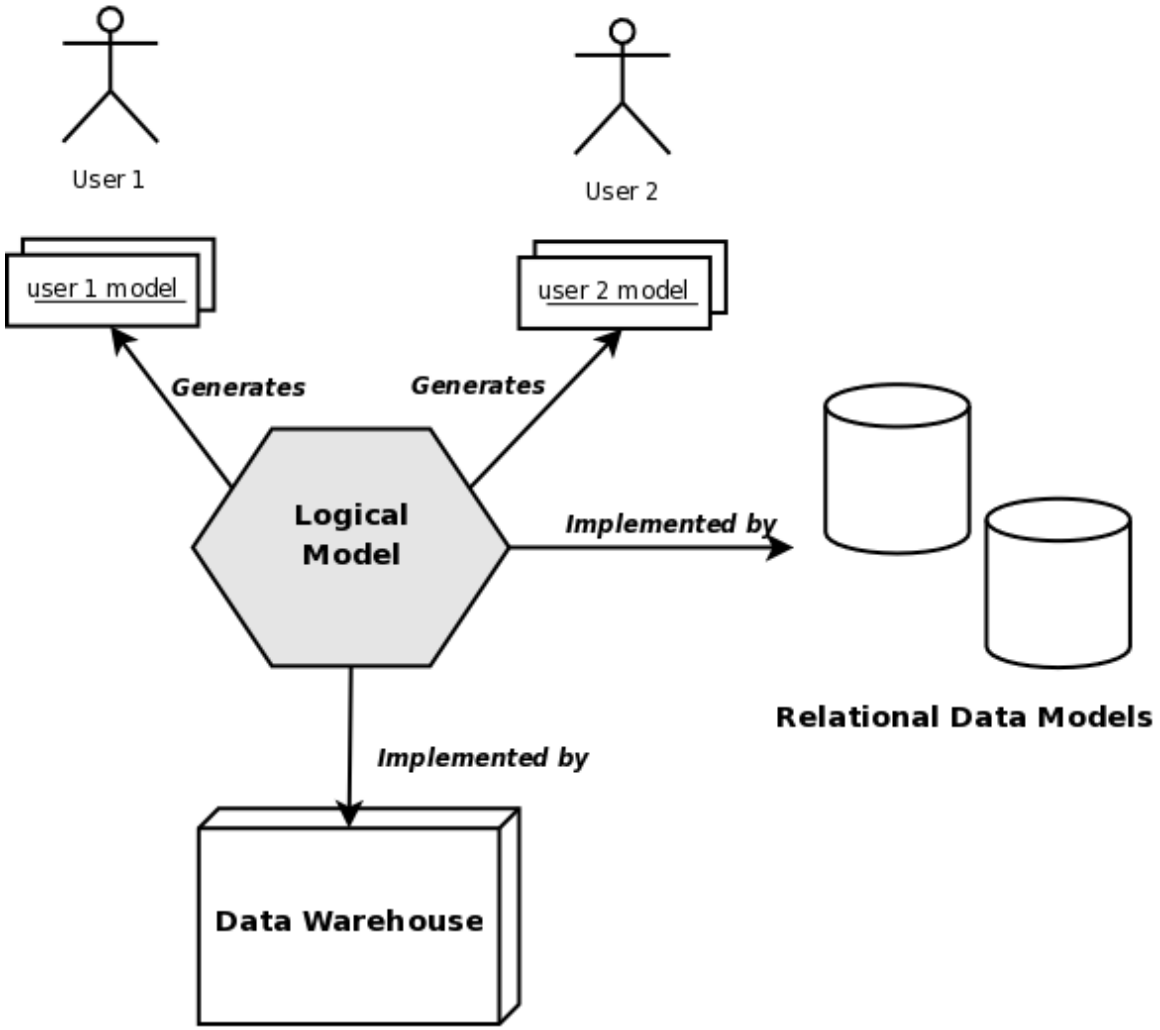
ANSI-SPARC 3 Level Architecture

- Formulated in the early days of database development
  - An attempt to standardize a model of the different ways of doing persistence
- Defines three views
  - External views: The data as organized by groups of users
    - Different groups of users have different views of the data
  - Conceptual view: The defined logical schema of the data
    - Only one conceptual view – the data dictionary for the project
    - Should be able to generate each external view from the conceptual view
  - Internal view: How the data is organized internally in a storage system
    - Different use cases require different models.
    - For example, online transaction processing versus historical analytic analysis

ANSI-SPARC 3 Level Architecture



ANSI-SPARC 3 Level Architecture



Chen's Model

- This is also very close to an influential paper on data persistence
  - Peter Chen's paper on entity relationship modeling.
- He proposes:
  - In the study of a data model, we should identify the levels of logical views of data with which the model is concerned
  - We can identify four levels of views of data:
    - Information concerning entities and relationships which exist in our minds. (external view)
    - Information structure – organization of information in which entities and relationships are represented by data. (conceptual view)
    - Access path independent data structure – the data structures which are not involved with search schemes, indexing schemes, etc. (internal view)
  - Access path dependent data structure (physical view)

The Fundamental Problem

- At the application level, we work with concepts at the conceptual level
  - Recall from the OO section, these are domain concepts
  - They have an iconic representation in the interfaces the users interact with
- At the implementation level, we work with the relational and other models
  - For example, the domain concepts are stored in tables in a relational data base
- The problem is mapping the domain objects in the interface to the actual storage mechanisms in the infrastructure
- The other problem is making this mapping loosely coupled enough so that
  - We can change the underlying physical data storage without breaking the client code
  - For example, we might want to migrate from MongoDB to PostgreSQL without having to change any client code

Coupling and Suppleness

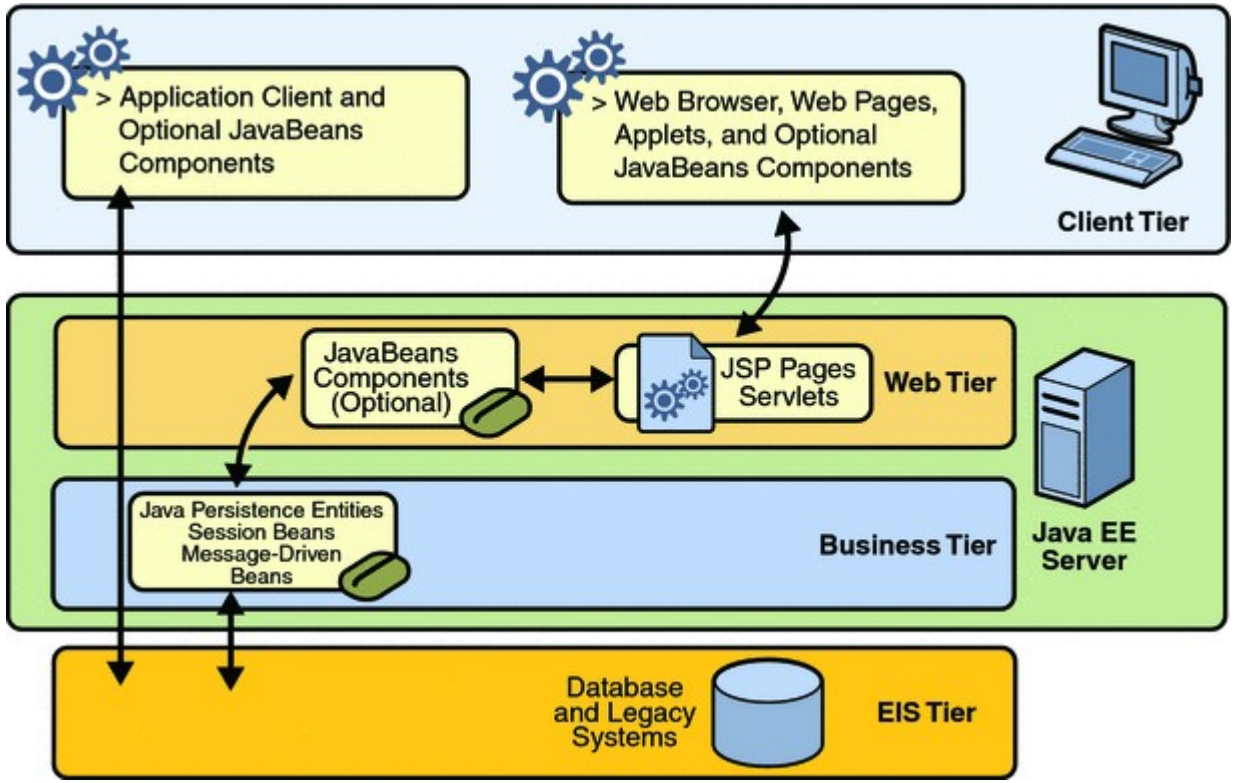
- The problem was that the Java data objects had to be mapped to the underlying relational database
  - Because the focus was on Java because at the time, most of the applications that had to access legacy databases were written in Java
- This was originally handled by JDBC code
  - Tried to create a layer of abstraction between the actual database and the Java code
  - But the Java code had to execute SQL statements and interpret the result
  - The resulting code was often brittle and tightly coupled to the database
- Changes to the underlying relational model could break a lot of Java code
  - For example, changing the name of a column in a table

Coupling and Suppleness

- Coupling the application to the implementation layer is a common mistake
  - The client now depends on the implementation, a specific relational model for example
  - If the relational model is changed, the client may break
- There are other forms of implementation models
  - We may want to migrate to a different type of implementation for performance or scaling reasons without having to rewrite the client
- We solve this by introducing a layer of indirection
  - In Microservices, this is persistence backing service
  - The backing service gets CRUD requests for domain objects from the client
  - Then maps to and from the corresponding underlying implementation
  - Data is received from the client and returned in a form that is implementation independent

The ORM Problem

- Originally relational data was the only game in town and applications had to connect to existing corporate data centers



Example from Oracle Docs

```
public static void viewTable(Connection con) throws SQLException {
  String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";
  try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
      String coffeeName = rs.getString("COF_NAME");
      int supplierID = rs.getInt("SUP_ID");
      float price = rs.getFloat("PRICE");
      int sales = rs.getInt("SALES");
      int total = rs.getInt("TOTAL");
      System.out.println(coffeeName + ", " + supplierID + ", " + price +
                         ", " + sales + ", " + total);
    }
  } catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
  }
}
```

J2EE Entity Beans

- The alternative to directly accessing the database from Java was implemented in J2EE as “Entity Beans”

```
import javax.ejb.*;
import java.rmi.*;

public interface EmployeeLocalHome extends EJBLocalHome
{

  public EmployeeLocal create(Integer empNo) throws CreateException;

  // Find an existing employee
  public EmployeeLocal findByPrimaryKey (Integer empNo) throws FinderException;

  //Find all employees
  public Collection findAll() throws FinderException;

  //Calculate the Salaries of all employees
  public float calcSalary() throws Exception;
}
```

J2EE Entity Beans

- The underlying database representation was not required in the Java code
  - Instead, it was moved into XML configuration files
  - These became very difficult to work with

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.EmployeeLocalHome</local-home>
         <local>employee.EmployeeLocal</local>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Employee</abstract-schema-name>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
        <primkey-field>empNo</primkey-field>
      </entity>
...
</enterprise-beans>
```

Persistence Interfaces

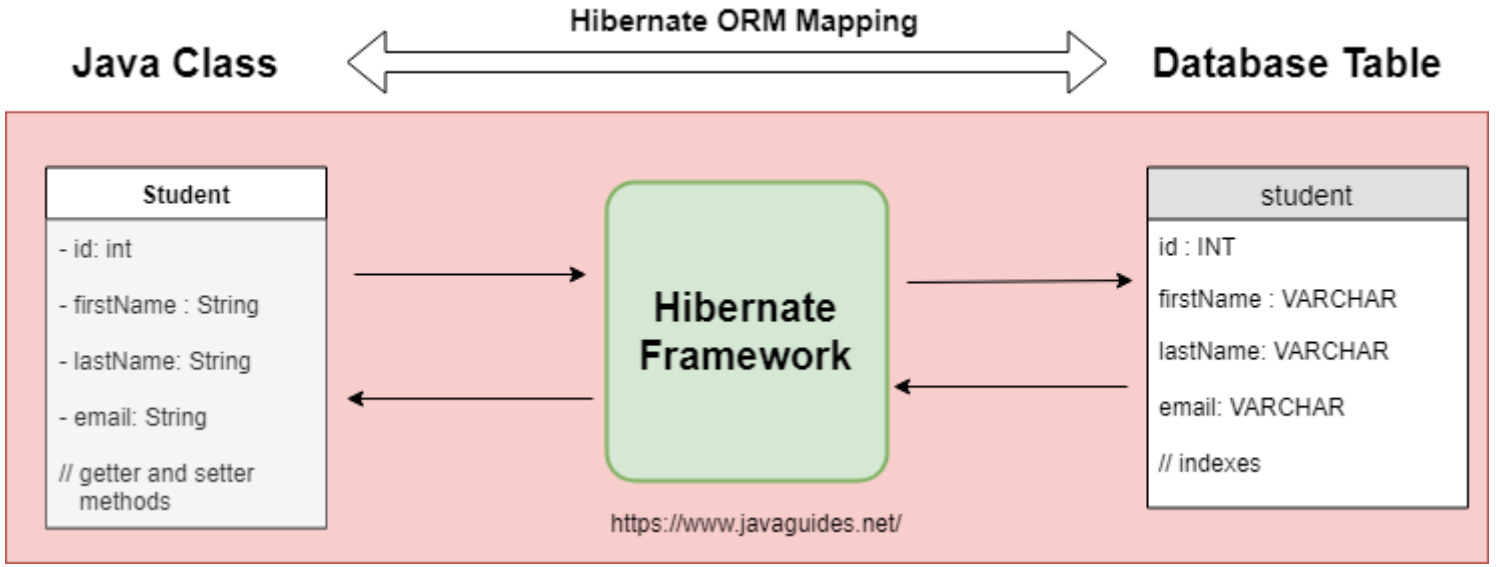
- Eventually, the solution evolved to:
  - Create a persistence interface that a client would call
  - The persistence entities would be domain objects
  - The interface is a backing service – follows the Dependency Inversion Principle
  - Provides the CRUD and other functionality required by the domain
- Implementation
  - For a specific repositories in the infrastructure, what ever type
  - There is an implementation that does the actual mapping of the domain object to the type of structure the data storage system requires

The JPA Standard

- By the time EJB 3.0 came around the JPA specification had been released
  - Like the rest of the EE specifications, it defined an interface
  - The interface standardizes how Java interacts with persistent data
  - Utilizes the concept of an “entity”
  - Abstracts out the general concept of a query to be independent of the underlying database
- The intent was to decouple the way the code referred to persistent objects from how they were actually implemented
  - The class that implements the interface does the work of mapping the data
  - If a different type of data persistence is used
  - Then the client code is kept the same, talking to the JPA interface
  - But the class that implements the interface is changed.

The JPA Standard

- Like other specifications, JPA defines an interface
  - This is implemented in various ORM products
  - Hibernate is a popular implementation



JPA Architecture and Interfaces

- JPA is made up of several interfaces
  - These define how code interacts with the persistence layer
  - The persistence layer is where the code that actually interacts with the database lives
  - The main interfaces are summarized in the table below

Interface	Description
`EntityManager`	Main interface to interact with persistence
`EntityManagerFactory`	Factory to create `EntityManager`
`EntityTransaction`	Manages transaction boundaries
`Query` / `TypedQuery`	Execute queries (JPQL / SQL)
`Persistence`	Entry point to JPA setup

EntityManager

- EntityManager is the main interface provided by the Java Persistence API (JPA) to
  - Allows code to interact with a database using Java objects instead of SQL.
  - Acts as translation layer between a Java application and a database.
- Provides basic CRUD functionality
  - Create new records (entities)
  - Retrieve records
  - Update records
  - Delete records
- Run queries using JPQL (Java Persistence Query Language)
  - JPQL works with entity classes and their fields, not table names or columns.
  - Decouples the logic of the query with the physical layout of the database
  - JPQL is database-agnostic.

EntityManagerFactory

- Represents a thread-safe, heavyweight object that is responsible for:
  - Creating and managing EntityManager instances
  - Holding database configuration and metadata
  - Caching entity mappings
  - Managing the underlying connection pool (via the JPA provider)
- One EntityManagerFactory is created per application
  - It must be closed explicitly when the app shuts down
  - Expensive to create so only created once

EntityManagerFactory Responsibilities

- Reads the persistence.xml File
  - This contains the mapping from classes to database tables
    - Database connection info (JDBC URL, driver, username, etc.)
    - Entity class declarations
    - JPA provider settings (e.g., Hibernate-specific properties)
- Loads the JPA ORM Provider
  - JPA delegates to the provider (e.g., Hibernate, EclipseLink).
  - The provider implements the low-level persistence logic.
- Parses and validates @Entity classes
  - For all entity classes found in the code are:
    - Scanned for annotations like @Entity, @Id, @OneToMany, etc.
    - Validated (e.g., checking if primary keys are defined)
    - Mapped to corresponding database tables
- Establishes and maintains database connections

@Entity

- @Entity is a marker annotation
  - Classes annotated with @Entity are treated as persistent entities
  - Means this is a Java class whose instances are stored as rows in a database table.
- JPA registers the class as an entity during application startup.
- Expects a corresponding table in the database
  - May be configured to create one if it doesn't exist
  - Manages the class's instances using an object-relational mapping (ORM).
- Other annotations
  - @Id - Marks id as the primary key
  - @GeneratedValue - Tells JPA to auto-generate the ID value

```
@Entity
public class Employee {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    private String department;
}
```

@Entity Classes

- In order to facilitate the ORM mapping, entity classes have to meet certain requirements
  - No-arg constructor - Must have a public or protected no-argument constructor
  - Unique identifier - Must have a field or property annotated with @Id
  - Not final - The class must not be final
  - Not abstract - Must be concrete if used directly
  - Serializable (optional) - Often recommended, especially in distributed apps

Spring Data JPA

- Spring Data JPA is a module of the Spring Data project
  - Integrates JPA into the Spring Framework
  - Eliminates most boilerplate JPA code (e.g., EntityManager usage)
  - Provides repository interfaces for CRUD and custom queries
  - Supports integration with Spring Boot for easy setup
- Provides a basic CRUDRepository Interface
  - Can be extended to add specialized methods

```
save(entity)          // Create or update
findById(id)          // Get by primary key
existsById(id)        // Check existence
findAll()             // Get all records
delete(entity)        // Delete by entity
deleteById(id)        // Delete by ID
```

Spring Boot Application Class

- Spring Boot automatically:
  - Detects @Entity classes
  - Scans and wires repository interfaces
  - Configures the JPA provider (e.g., Hibernate)
  - Loads the database connection from application.properties
- We will be using Spring Data in the lab
  - This simplifies the code you will have to work with

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

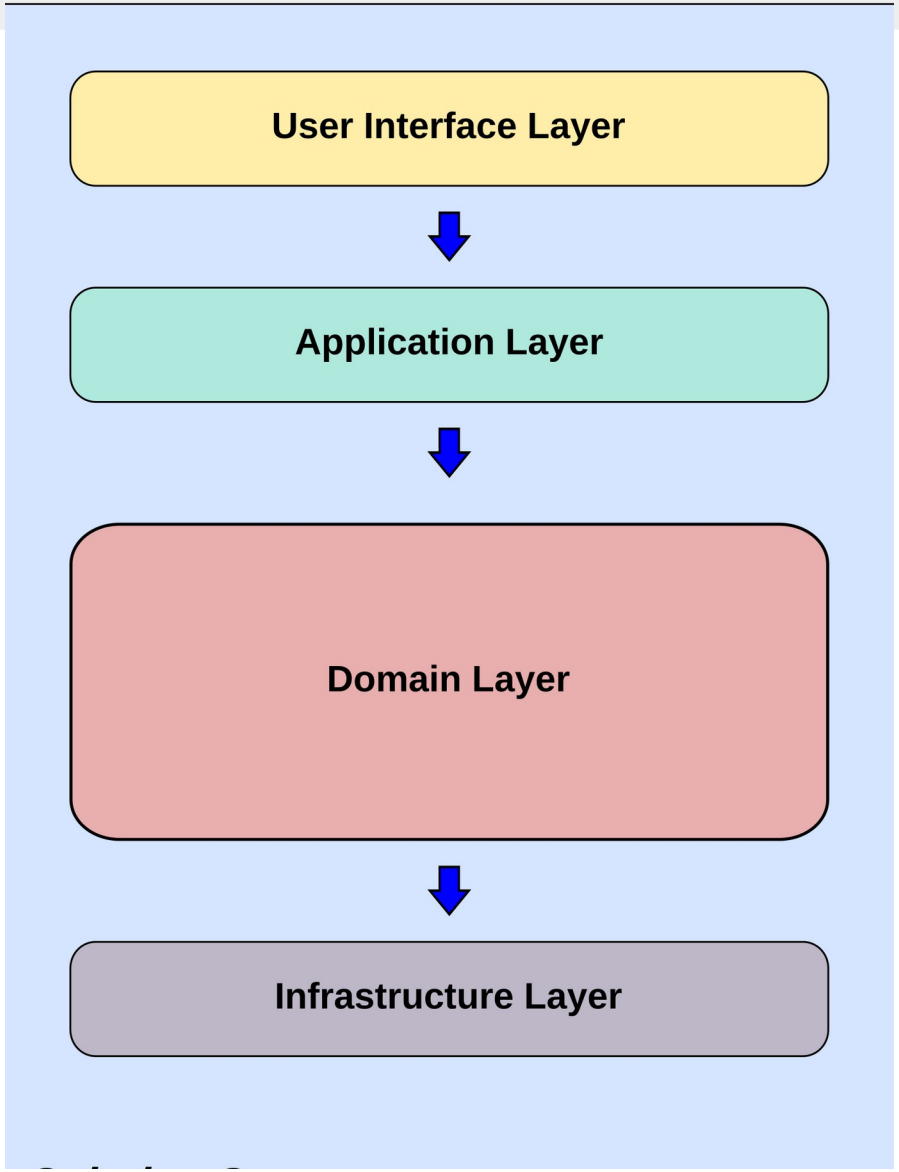
Working with Schemas

- Schemas are a structured defined representation of data
- Corresponds to the business entities in a context
- We prefer to work with schemas for inter-module communication
  - These correspond to logical models
  - We leave the implementation mappings up to the infrastructure services

Layering

In modern architectures

- Specifically microservices
- The application layer represents a process (“Buy this book online”)
- The domain layer does the work
  - An account service that handles updating the account
  - A shipping service that handles fulfilling the order, and so on
- The infrastructure layer is where persistence occurs
  - It provides an interface the module in the domain layer can call to get persistence services
  - Eg, Get the account details for a customer
  - Expressed in terms of schemas



Working with Schemas

- Since each component may have its own logical model, moving data between contexts may be a problem
- We prefer to work with schemas for inter-module communication
  - These correspond to logical models
  - We leave the implementation mappings up to the infrastructure services

US Subsidiary

```
{
  person: {
    firstName: string
    lastName: string
    dob: date
  },
  address: {
    address_1: string
    address_2: string
    city: string
    state: string
    zip: string
  }
}
```

Canadian Subsidiary

```
{
  person: {
    givenName: string
    surName: string
    birthday: date
  },
  address: {
    address_1: string
    address_2: string
    city: string
    province: string
    postal_code: string
  }
}
```

Schema Registry

- The one source of truth
  - A master data model corresponding to the conceptual model
- Documents how known schemas are converted from one to another
  - The schema in each component can the thought of as representing a view of the data
- Often exist because of legacy systems
  - When they are accessed, we still need to manage legacy data in the format the legacy system knows
- Also occurs when connecting previously independent systems
  - Merging an American company with a Japanese company for example

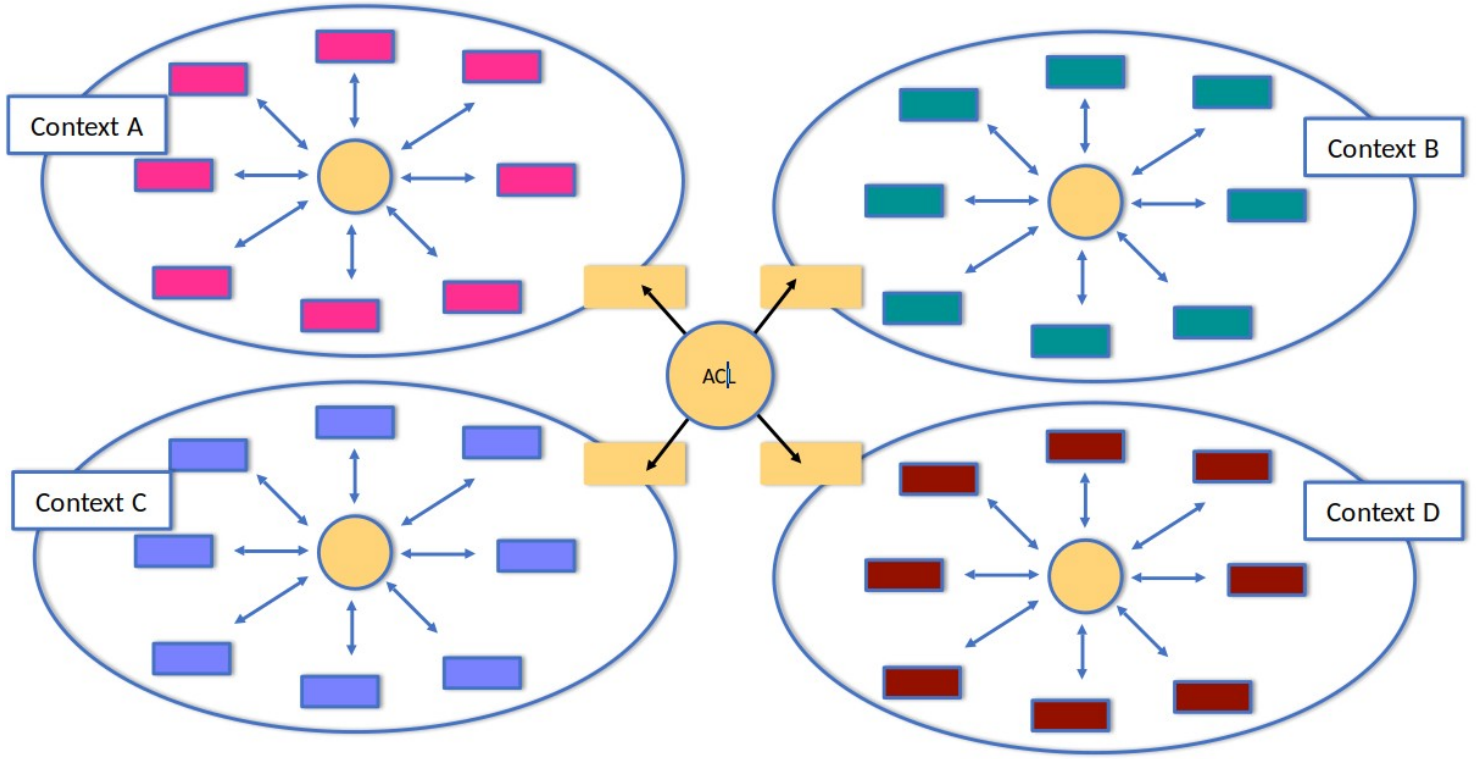
Schema Registry

- The one source of truth
  - A master data model corresponding to the conceptual model
- Documents how known schema are converted from one to another
  - The schema in each component can the thought of a representing a view of the data
- Often exist because of legacy systems
  - When they are accessed, we still need to manage legacy data in the format the legacy system knows
- Also occurs when connecting previously independent systems
  - Merging an American company with a Japanese company for example

Schema Registry

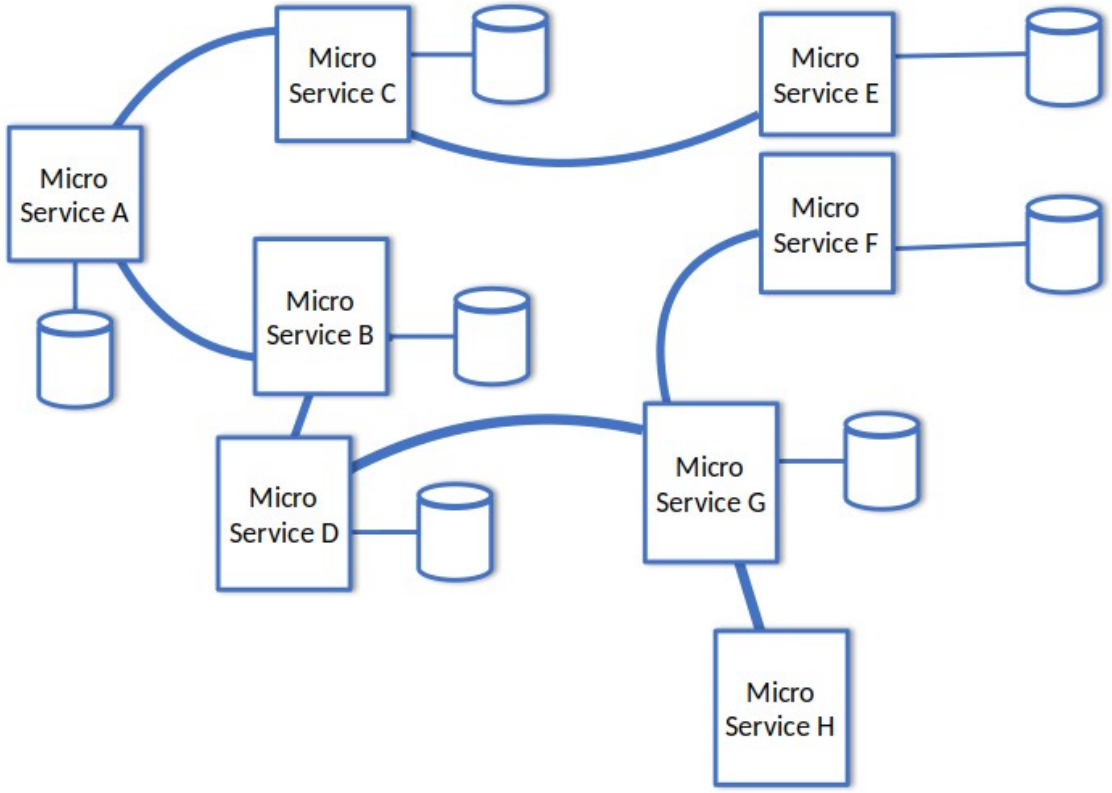
Anti-corruption Layer

- A service that intercepts data between contexts and translates from one schema to another
  - Useful when there is no schema registry



Distributed Data and Microservices

- Ideally, each microservice has its own view of the data and potentially its own repository



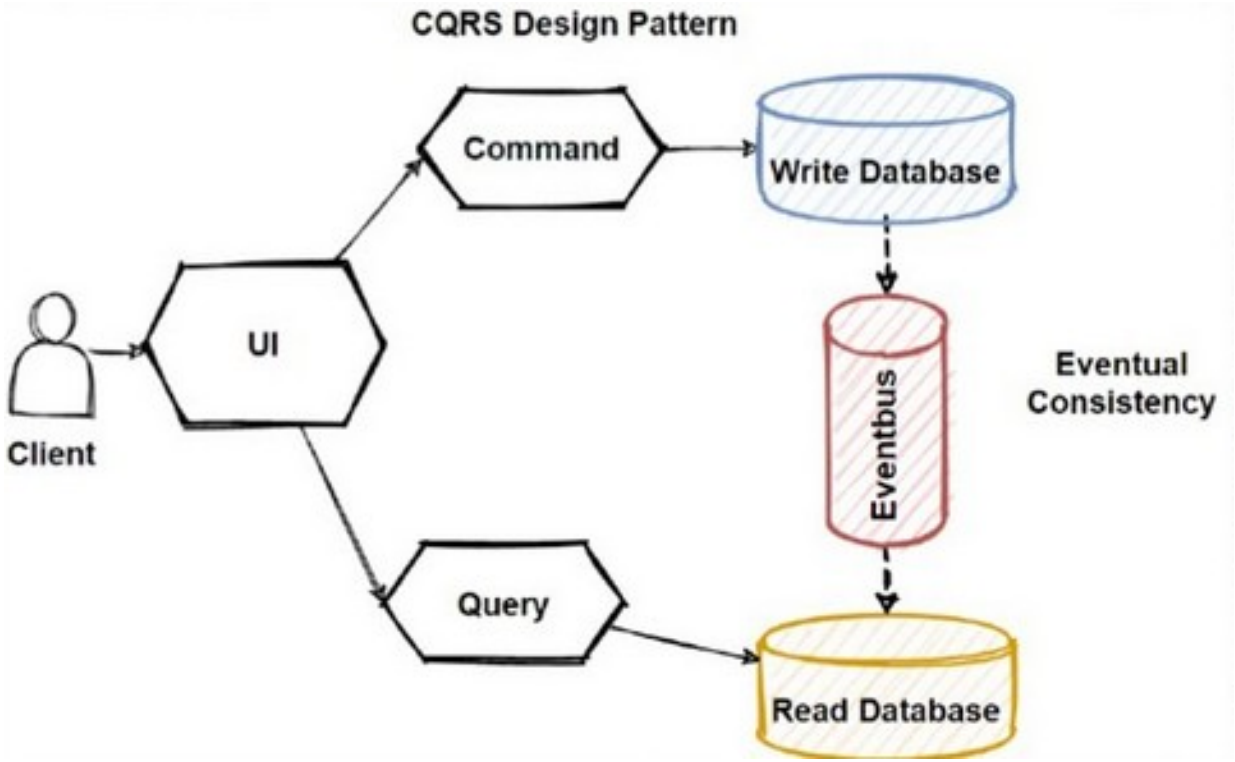
The Segmentation Problem

- Microservices may share a common repository
  - Or they may use local copies of the same data or different subsets of the same data
  - In this case we have distributed data
- For example
  - An order service creates an order, records it in the local database
  - The order is sent to the fulfillment service - there are now two copies of the order
- The so-called CAP theorem describes this situation.

The Segmentation Problem

Also occurs when we use command-query segregation

- This is a pattern to even out load on a database
- Since 80% of traffic is often read requests
- One database is where writes occur
- And multiple read only copies of the database
- The write database has to propagate changes to the read databases to keep them up to date



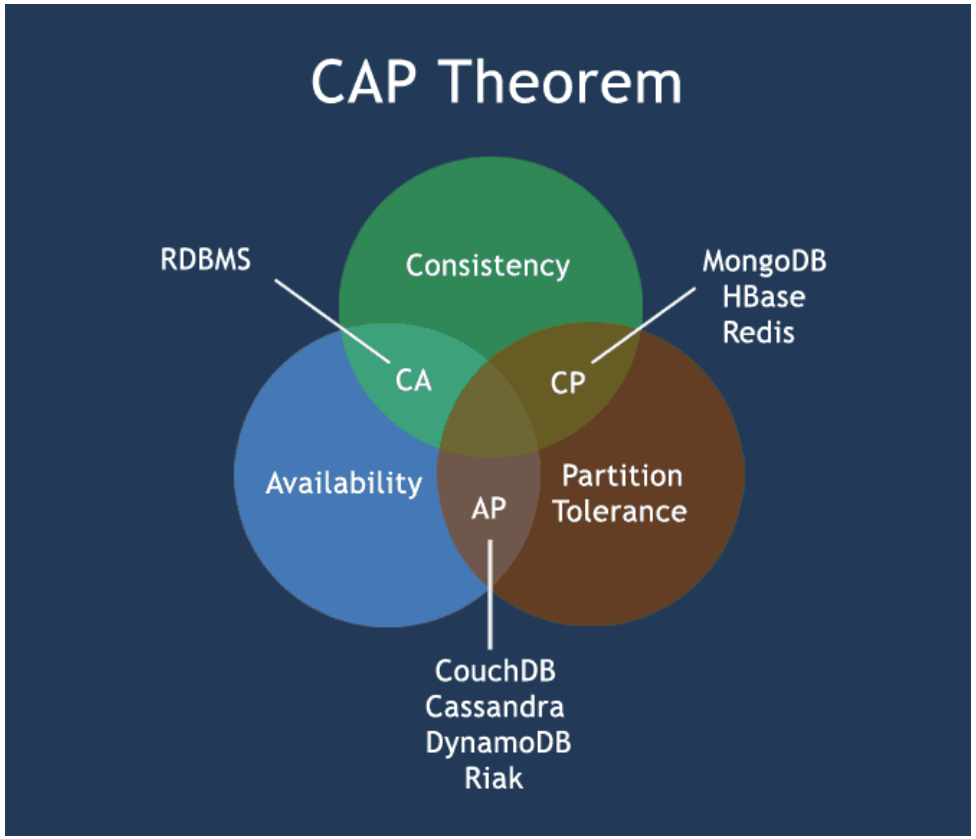
Distributed Data CAP Theorem

When data is distributed

- It is impossible to have both
- Consistency: The data is the same across every copy
- Availability: The data in each copy is accessible now

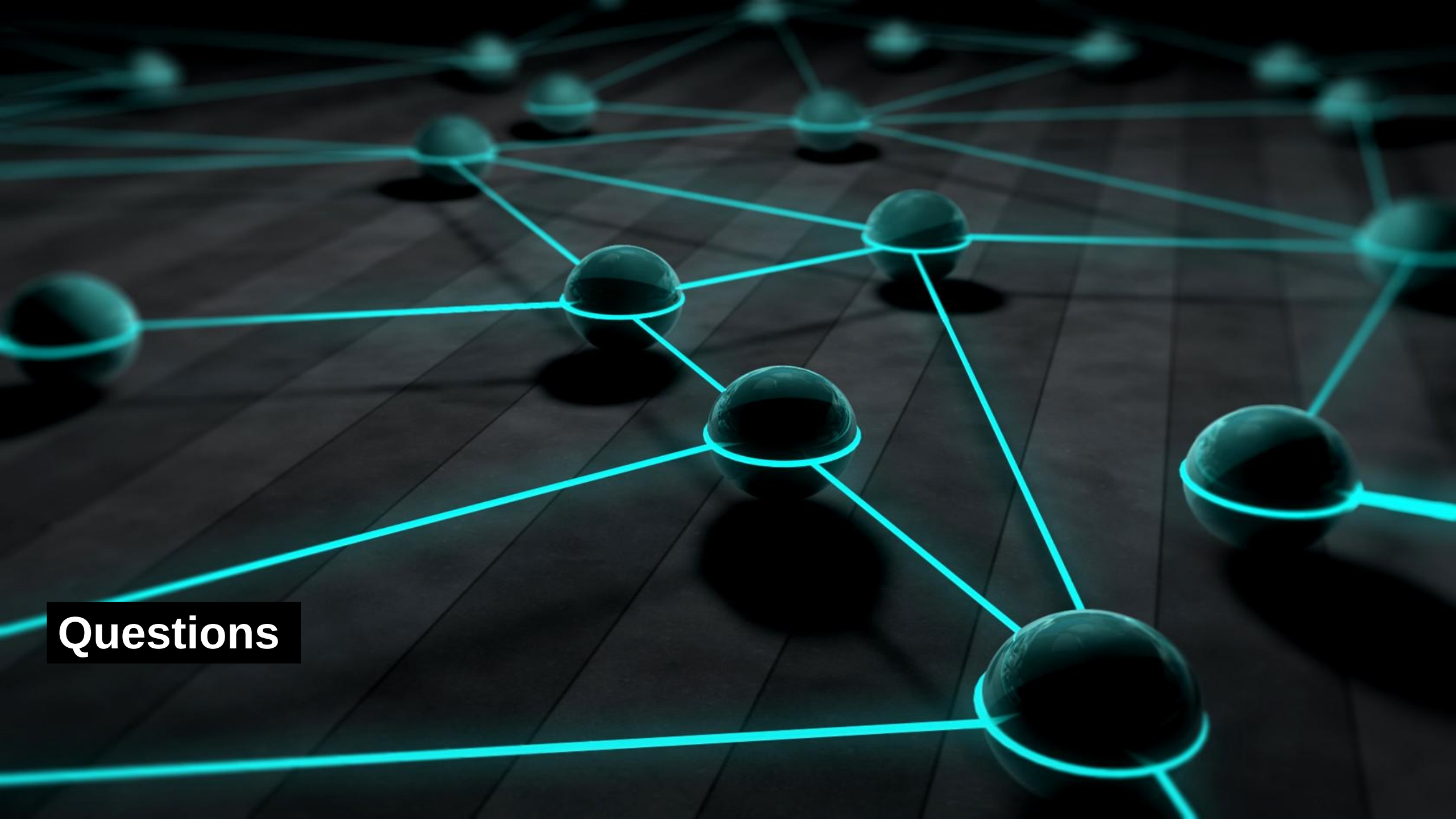
The problem

- If you want instant availability, the results will be inconsistent until the copies are all updated
- If you want consistency, some copies will not be available until their data is updated
- We generally choose balance between the two called consistency levels



Consistency Levels

		Consistency	Performance	Availability
Strong	See all previous writes.	excellent	poor	poor
Consistent Prefix	See initial sequence of writes.	okay	good	excellent
Bounded Staleness	See all “old” writes	good	okay	poor
Monotonic Read	See increasing subset of writes	okay	good	good
Read Your Writes	See all writes performed by reader	okay	okay	okay
Eventual	See subset of previous writes.	poor	excellent	excellent


Questions