Relational Databases

CS 240 – Advanced Programming Concepts

Database Management Systems (DBMS)

- Databases are implemented by software systems called Database Management Systems (DBMS)

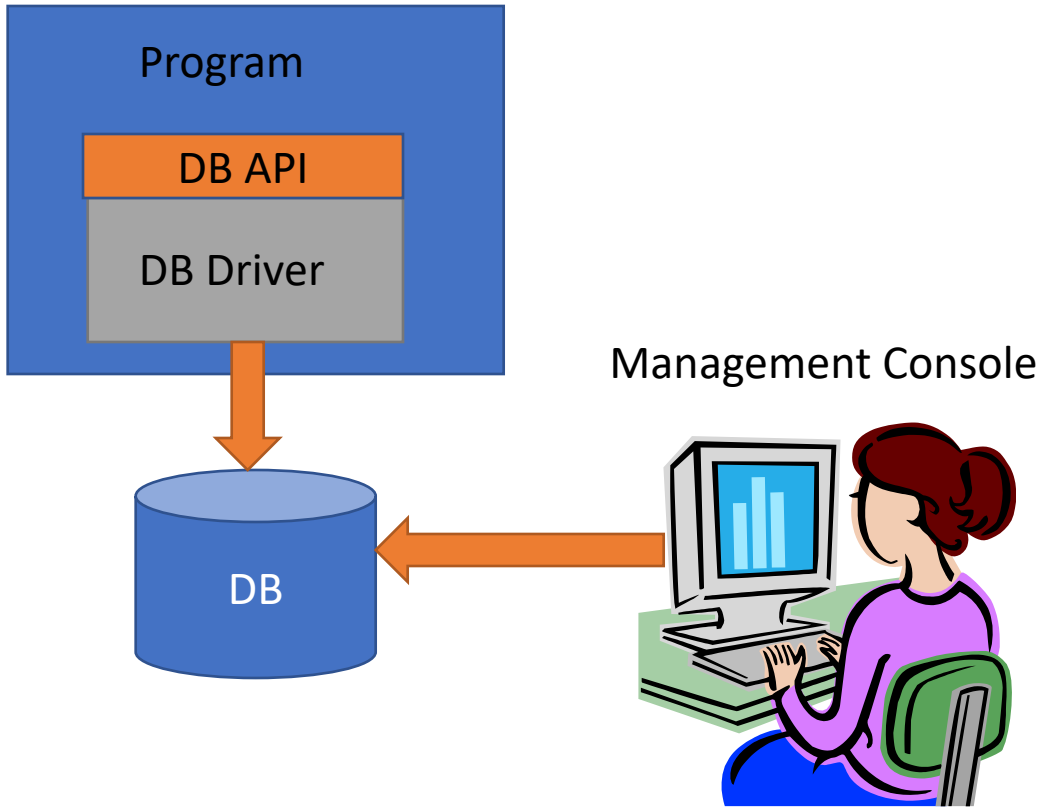
- Commonly used Relational DBMS's include Oracle, MySQL, PostgreSQL, and MS SQL Server

- DBMS's store data in files in a way that scales to large amounts of data and allows data to be accessed efficiently

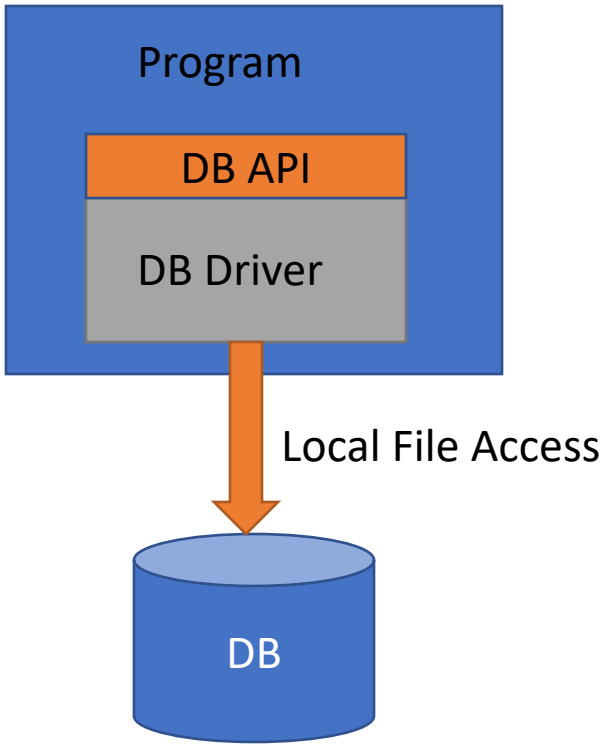
Programmatic vs. Interactive Database Access

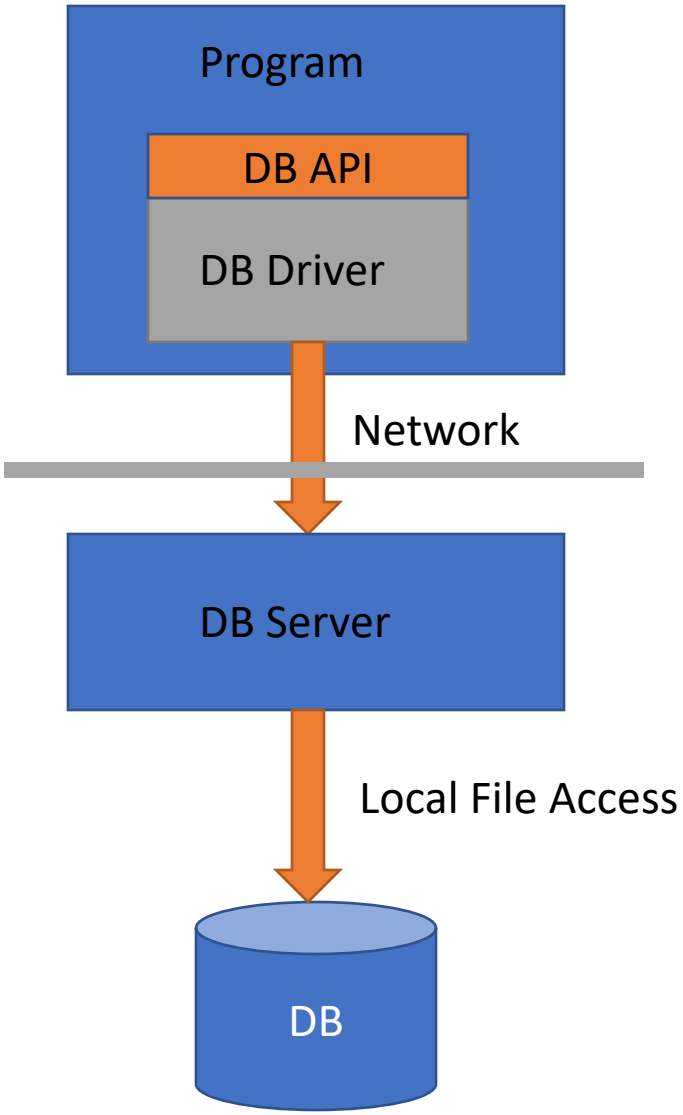
Programs can access a database through APIs such as JDBC or ADO.NET.

End users can access a database through an interactive management application that allows them to query and modify the database.



Embedded vs. Client/Server





Some DBMS's are Embedded only.
Some are Client/Server only.
Some can work in either mode.

Relational Databases

- Relational databases use the relational data model you learned about in CS 236
  - Relations = Tables
  - Tuples = Rows
- In the relational data model, data is stored in tables consisting of columns and rows.
  - Tables are like classes
  - Each row in a table stores the data you may think of as belonging to an object.
  - Columns in a row store the object's attributes (instance variables).
- Each row has a "Primary Key" which is a unique identifier for that row. Relationships between rows in different tables are represented using keys.
  - The primary key in one table matches a foreign key in another table
- Taken together, all the table definitions in a database make up the "schema" for the database.

Comparison of Object and Relational Models

Object Model

- Class
- Object
- Relationship (reference)

Relational Data Model

- Table
- Row
- Relationship (primary and foreign key)

Book Club Schema

member

id	name	email_address
1	‘Ann’	‘ann@cs.byu.edu’
2	‘Bob’	‘bob@cs.byu.edu’
3	‘Chris’	‘chris@cs.byu.edu’

books_read

member_id	book_id
1	1
1	2
2	2
2	3
3	3
3	4

book

id	title	author	genre	category_id
1	‘Decision Points’	‘George W. Bush’	‘NonFiction’	7
2	‘The Work and the Glory’	‘Gerald Lund’	‘HistoricalFiction’	3
3	‘Dracula’	‘Bram Stoker’	‘Fiction’	8
4	‘The Holy Bible’	‘The Lord’	‘NonFiction’	5

genre

genre	description
‘NonFiction’	…
‘Fiction’	…
‘Historical Fiction’	…

Book Club Schema

category

id	name	parent_id
1	‘Top’	Null
2	‘Must Read’	1
3	‘Must Read (New)’	2
4	‘Must Read (Old)’	2
5	‘Must Read (Really Old)’	2
6	‘Optional’	1
7	‘Optional (New)’	6
8	‘Optional (Old)’	6
9	‘Optional (Really Old)’	6

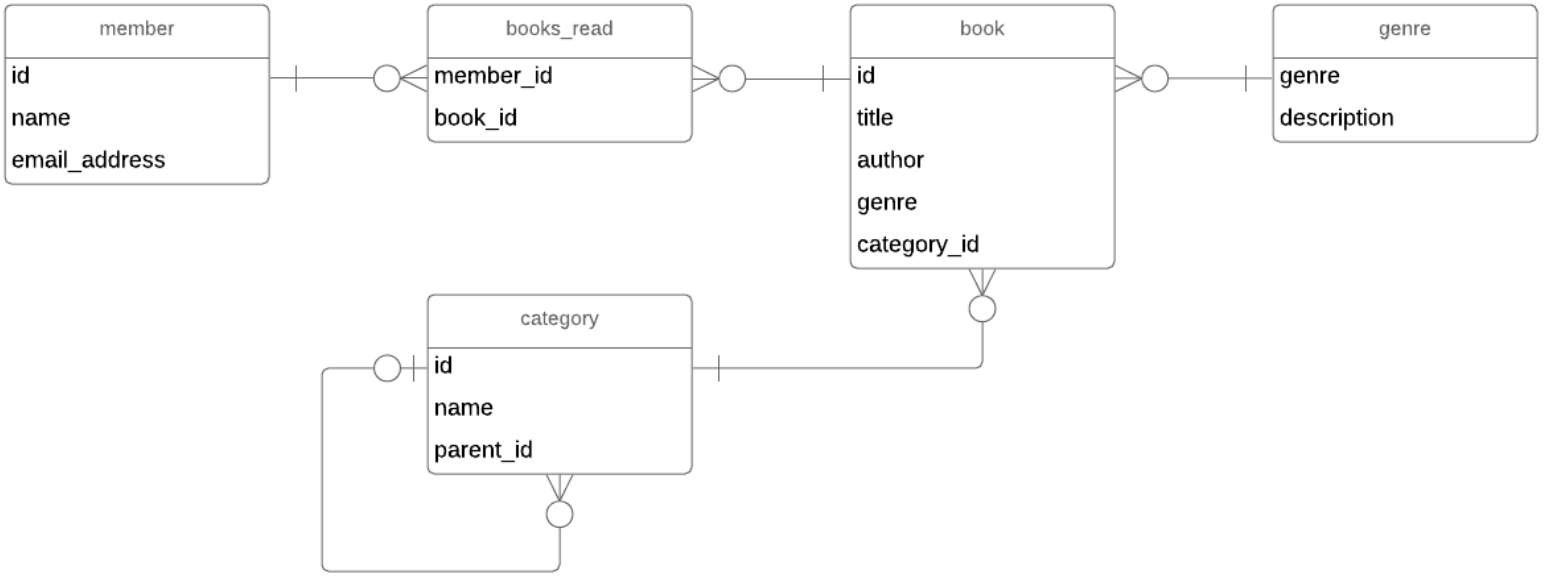
Modeling Object Relationships

- Connections between objects are represented using foreign keys
- Foreign Key: A column in table T_1 stores primary keys of rows in table T_2
- Book Club Examples
  - Books_Read table stores Member and Book keys
  - Book table stores Category key
  - Category table stores parent Category key

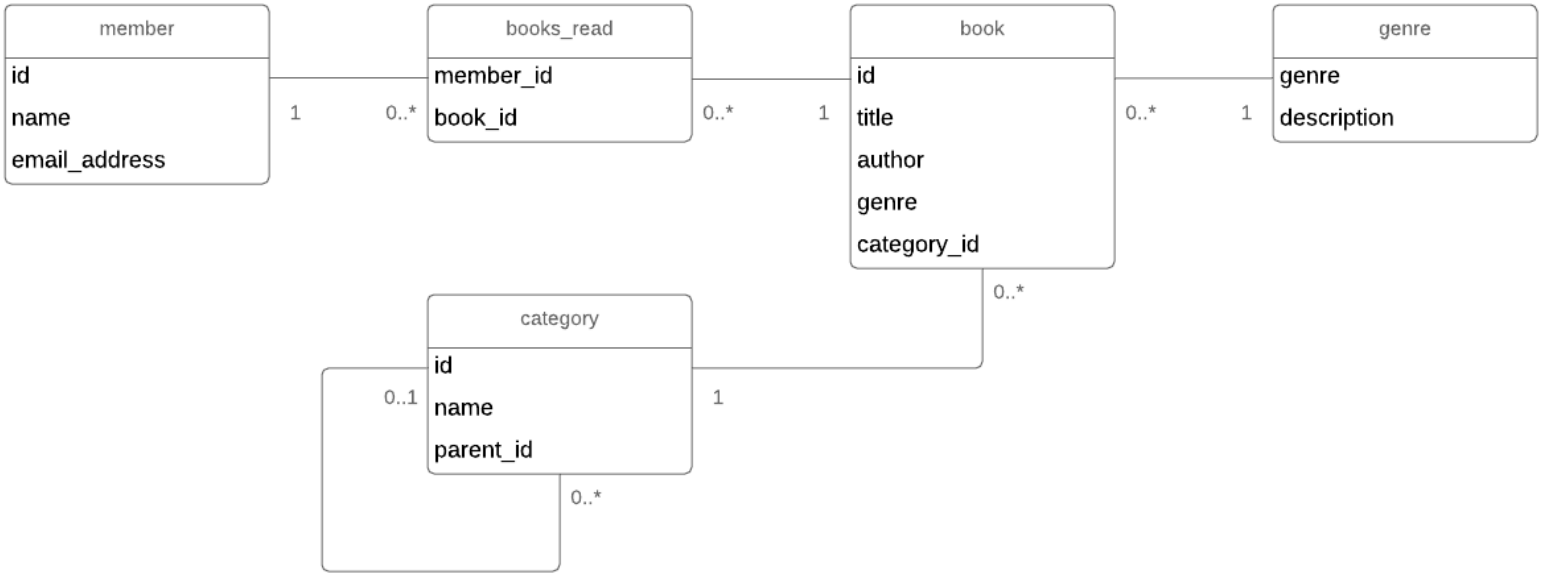
Modeling Object Relationships

- Types of Object Relationships
  - One-to-One
    - A Person has one Head; A Head belongs to one Person
    - Either table contains a foreign key referencing the other table
  - One-to-Many
    - A Book has one Category; a Category has many Books
    - A Book has one Genre; a Genre has many Books
    - A Category has one parent Category; a Category has many sub Categories
    - The “Many” table contains a foreign key referencing the “One” table
  - Many-to-Many
    - A Member has read many Books; a Book has been read by many Members
    - Create a “join table” (sometimes called an intersecting entity) whose rows contain foreign keys of both tables

Book Club Entity Relationship Diagram (ERD)

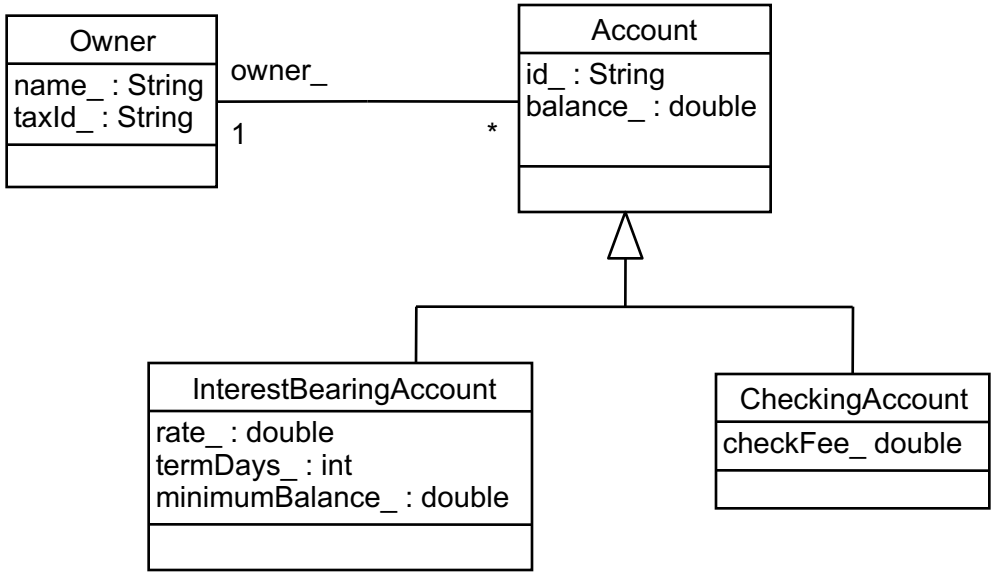


Book Club Database Model (UML)



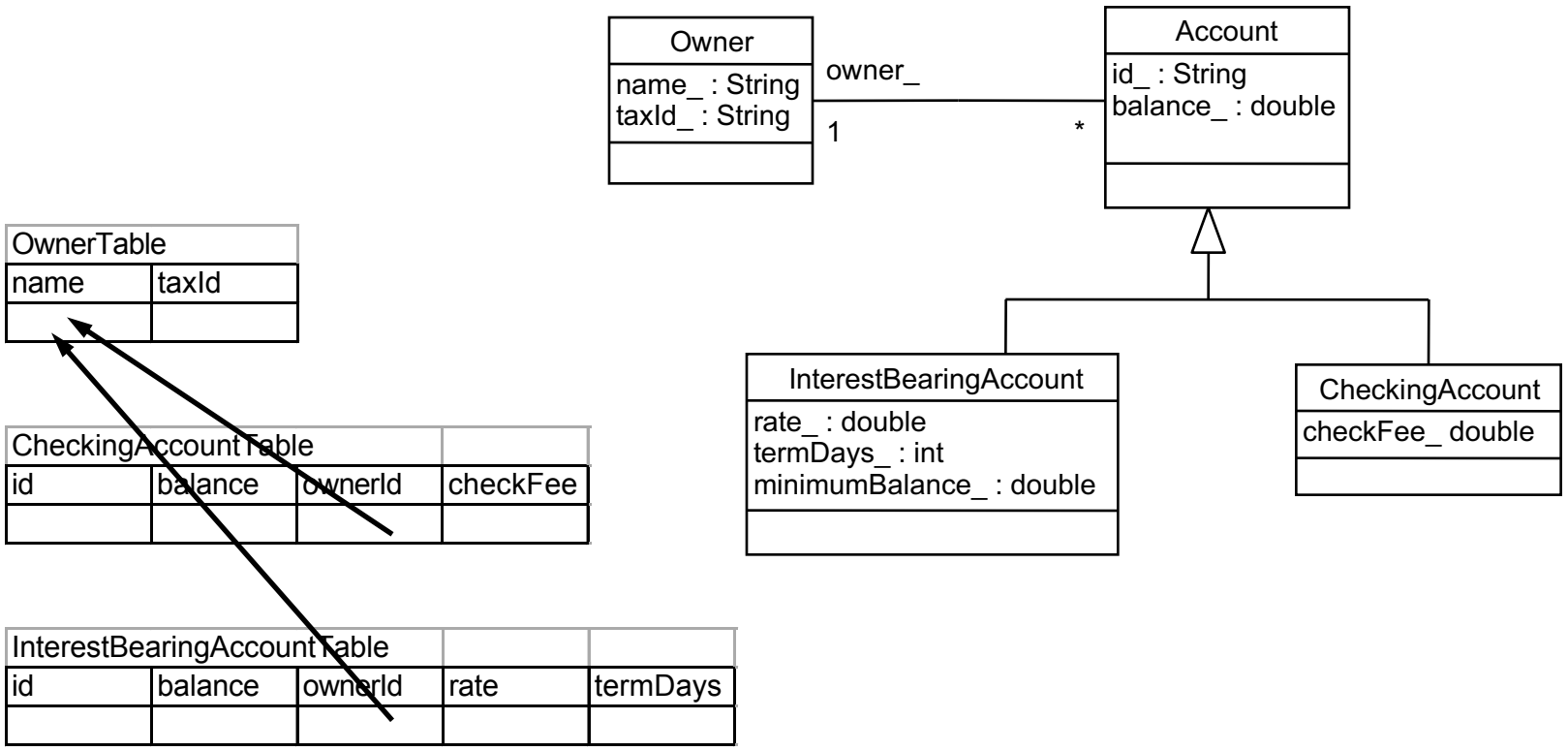
Modeling Inheritance Relationships

- How do we map the following Class Model to an RDBMS



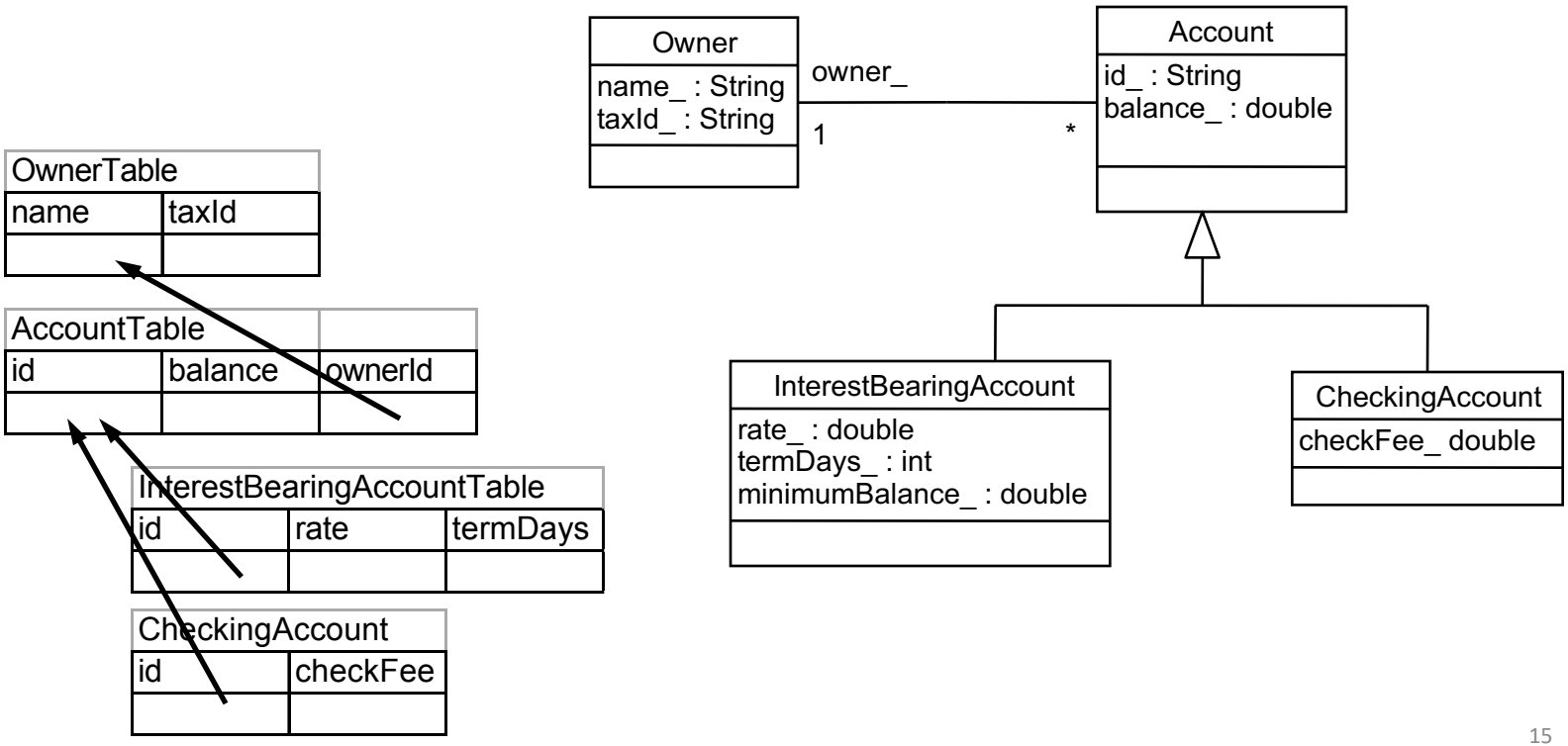
Horizontal Partitioning

- Each concrete class is mapped to a table



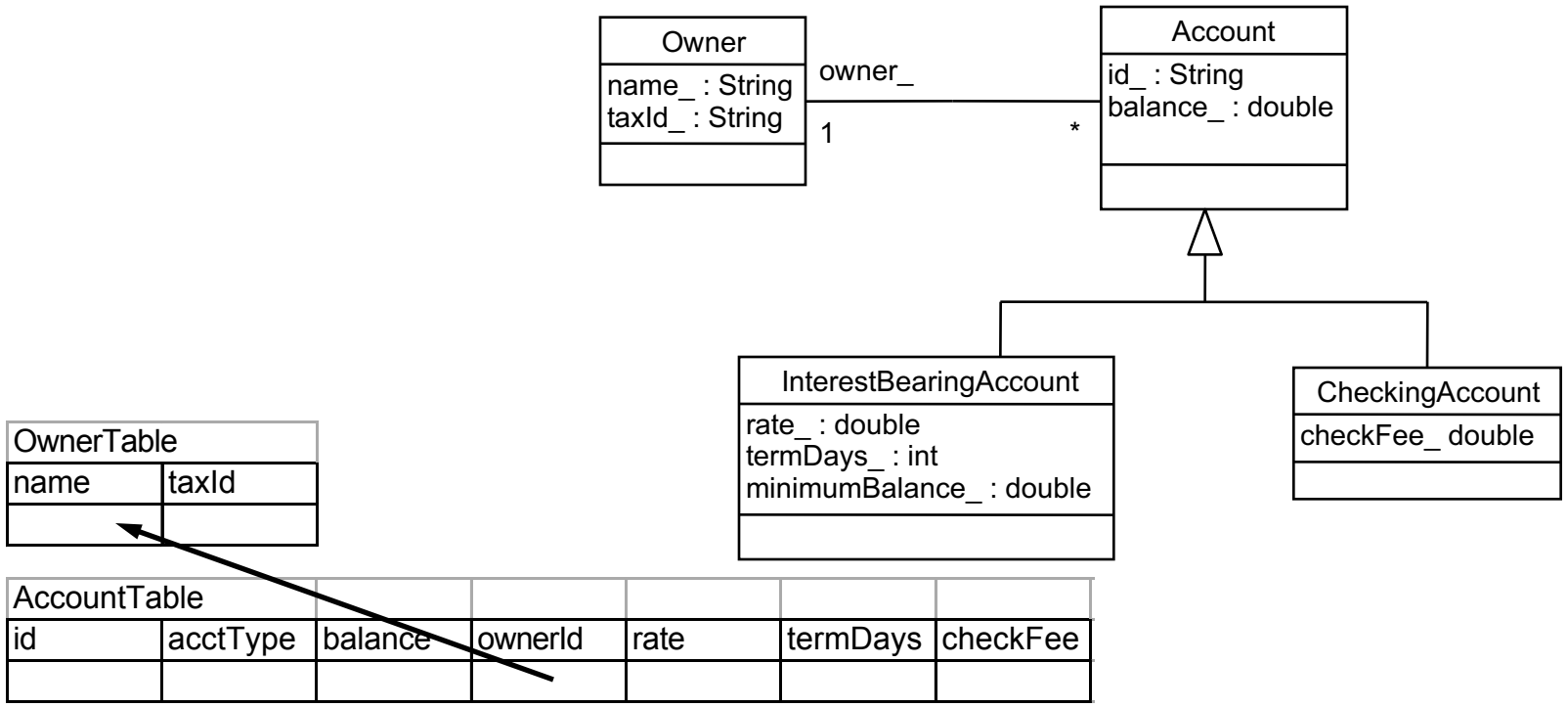
Vertical Partitioning

- Each class is mapped to a table



Unification

- Each sub-class is mapped to the same table



RDBMS Mapping

- Horizontal Partitioning
  - Entire object within one table
  - Only one table required to activate object
  - No unnecessary fields in the table
  - Must search over multiple tables for common properties
- Vertical Partitioning
  - Object spread across different tables
  - Must join several tables to activate object
- Vertical Partitioning (cont.)
  - No unnecessary fields in each table
  - Only need to search over parent tables for common properties
- Unification
  - Entire object within one table
  - Only one table required to activate object
  - Unnecessary (blank) fields in the table
  - All sub-types will be located in a search of the common table

Structured Query Language (SQL)

SQL

- Language for performing relational database operations
  - Create tables
  - Delete tables
  - Insert rows
  - Update rows
  - Delete rows
  - Query for matching rows
  - Much more …

SQL Data Types – Strings

- Each column in an SQL table declares the type that column may contain
- **Character strings**
  - CHARACTER(*n*) or CHAR(*n*) — fixed-width *n*-character string, padded with spaces as needed
  - CHARACTER VARYING(*n*) or VARCHAR(*n*) — variable-width string with a maximum size of *n* characters
- **Bit strings**
  - BIT(*n*) — an array of *n* bits
  - BIT VARYING(*n*) — an array of up to *n* bits

SQL Data Types – Numbers and Large Objects

- **Numbers**
  - INTEGER and SMALLINT
  - FLOAT, REAL and DOUBLE PRECISION
  - NUMERIC(*precision*, *scale*) or DECIMAL(*precision*, *scale*)
- **Large objects**
  - BLOB – binary large object (images, sound, video, etc.)
  - CLOB – character large object (text documents)

SQL Data Types – Date and Time

- DATE — for date values (e.g., 2011-05-03)
- TIME — for time values (e.g., 15:51:36). The granularity of the time value is usually a *tick* (100 nanoseconds).
- TIME WITH TIME ZONE or TIMETZ — the same as TIME, but including details about the time zone.
- TIMESTAMP — This is a DATE and a TIME put together in one variable (e.g., 2011-05-03 15:51:36).
- TIMESTAMP WITH TIME ZONE or TIMESTAMPTZ — the same as TIMESTAMP, but including details about the time zone.

SQLite Data Types

- SQLite is a lightweight (simple) RDBMS we will use in this class
- SQLite stores all data using the following data types
  - INTEGER
  - REAL
  - TEXT
  - BLOB
- SQLite supports the standard SQL data types by mapping them onto the INTEGER, REAL, TEXT, and BLOB types

Creating Tables

- CREATE TABLE
  - Primary Keys
  - Null / Not Null
  - Autoincrement
  - Foreign keys

```
create table book
(
      id integer not null primary key autoincrement,
      title varchar(255) not null,
      author varchar(255) not null,
      genre varchar(32) not null,
      category_id integer not null,
      foreign key(genre) references genre(genre),
      foreign key(category_id) references category(id)
);
```

Foreign Key Constraints

- Not required – can query without them
- Enforce that values used as foreign keys exist in their parent tables
- Disallow deletes of the parent table row when referenced as a foreign key in another table
- Disallow updates of the parent row primary key value if that would "orphan" the foreign keys
- Can specify that deletes and/or updates to the primary keys automatically affect the foreign key rows
  - `foreign key(genre) references genre(genre) on update cascade on delete restrict`
- Available actions:
  - No Action, Restrict, Set Null, Set Default, Cascade

Dropping Tables

- Drop Table
  - `drop table book;`
  - `drop table if exists book;`

- When using foreign key constraints, order of deletes matters
  - Can't delete a table with values being used as foreign keys in another table (delete the table with the foreign keys first)

Inserting Data into Tables

- INSERT
  - ```
    insert into book
    (title, author, genre, category_id)
    values ('The Work and the Glory',
    'Gerald Lund', 'HistoricalFiction', 3);
    ```

- Complete Example
  - create-db.sql.txt
    ```

Updates

UPDATE Table
SET Column = Value, Column = Value, …
WHERE Condition

Change a member's information

```
UPDATE  member
SET name = 'Chris Jones',
    email_address = 'chris@gmail.com'
WHERE  id = 3
```

Set all member email addresses to empty

```
UPDATE  member
SET email_address = ''
```

Deletes

DELETE FROM Table
WHERE Condition

Delete a member

```
DELETE FROM member
WHERE  id = 3
```

Delete all readings for a member

```
DELETE FROM books_read
WHERE  member_id = 3
```

Delete all books

```
DELETE FROM book
```

Queries

```
SELECT  Column, Column, …
FROM  Table, Table, …
WHERE  Condition
```

Queries

book

id	title	author	genre	category_id
1	‘Decision Points’	‘George W. Bush’	‘NonFiction’	7
2	‘The Work and the Glory’	‘Gerald Lund’	‘HistoricalFiction’	3
3	‘Dracula’	‘Bram Stoker’	‘Fiction’	8
4	‘The Holy Bible’	‘The Lord’	‘NonFiction’	5

List all books

```
SELECT *
FROM book
```

result

id	title	author	genre	category_id
1	‘Decision Points’	‘George W. Bush’	‘NonFiction’	7
2	‘The Work and the Glory’	‘Gerald Lund’	‘HistoricalFiction’	3
3	‘Dracula’	‘Bram Stoker’	‘Fiction’	8
4	‘The Holy Bible’	‘The Lord’	‘NonFiction’	5

Queries

book

id	title	author	genre	category_id
1	‘Decision Points’	‘George W. Bush’	‘NonFiction’	7
2	‘The Work and the Glory’	‘Gerald Lund’	‘HistoricalFiction’	3
3	‘Dracula’	‘Bram Stoker’	‘Fiction’	8
4	‘The Holy Bible’	‘The Lord’	‘NonFiction’	5

List the authors and titles of all non-fiction books

```
SELECT author, title
FROM book
WHERE genre = ‘NonFiction’
```

result

author	title
‘George W. Bush’	‘Decision Points’
‘The Lord’	‘The Holy Bible’

Queries

category

id	name	parent_id
1	‘Top’	Null
2	‘Must Read’	1
3	‘Must Read (New)’	2
4	‘Must Read (Old)’	2
5	‘Must Read (Really Old)’	2
6	‘Optional’	1
7	‘Optional (New)’	6
8	‘Optional (Old)’	6
9	‘Optional (Really Old)’	6

List the sub-categories of category ‘Top’

```
SELECT id, name, parent_id
FROM category
WHERE parent_id = 1
```

result

id	name	parent_id
2	‘Must Read’	1
6	‘Optional’	1

Queries – Cartesian Product

List the books read by each member

```
SELECT member.name, book.title
FROM member, books_read, book
```

Member x Books_Read x Book **(3 x 6 x 4 = 72 rows)**

name	title
'Ann'	'Decision Points'
'Ann'	'The Work and the Glory'
'Ann'	'Dracula'
'Ann'	'The Holy Bible'
'Ann'	'Decision Points'
…	…
'Chris'	'The Holy Bible'

Probably not what you intended

Queries - Join

List the books read by each member

```
SELECT member.name,  book.title
FROM member, books_read,  book
WHERE member.id = books_read.member_id AND
      book.id = books_read.book_id
```

result

name	title
'Ann'	'Decision Points'
'Ann'	'The Work and the Glory'
'Bob'	'The Work and the Glory'
'Bob'	'Dracula'
'Chris'	'Dracula'
'Chris'	'The Holy Bible'

Database Transactions

- By default, each SQL statement is executed in a transaction by itself
- Transactions are useful when they consist of multiple SQL statements, since you want to make sure that either all of them or none of them succeed
- For a multi-statement transaction,
  - BEGIN TRANSACTION;
  - SQL statement 1;
  - SQL statement 2;
  - …
  - COMMIT TRANSACTION; or ROLLBACK TRANSACTION;

Java Database Access (JDBC)

Database Access from Java

- Load database driver
- Open a database connection
- Start a transaction
- Execute queries and/or updates
- Commit or Rollback the transaction
- Close the database connection

- Retrieving auto-increment ids

Omit transaction steps if you only need to execute a single statement.

Load Database Driver

```
try {
    // Legacy. Modern DB drivers don't require this
    Class.forName("org.sqlite.JDBC");
} catch(ClassNotFoundException e) {
    // ERROR! Could not load database driver
}
```

Open a Database Connection / Start a Transaction

```
import java.sql.*;

String dbName = "db" + File.separator + "bookclub.sqlite";
String connectionURL = "jdbc:sqlite:" + dbName;

Connection connection = null;
try {
    // Open a database connection
    connection = DriverManager.getConnection(connectionURL);

    // Start a transaction
    connection.setAutoCommit(false);
} catch (SQLException e) {
    // ERROR
}
```

Close the connection when you are through with it, or open it in a try-with-resources statement.

Don't close before you commit or rollback your transaction.

Execute a Query

```
PreparedStatement stmt = null;
ResultSet rs = null;
try {
    String sql = "select id, title, author, genre, " +
        " category_id  from book";
    stmt = connection.prepareStatement(sql);

    rs = stmt.executeQuery();
    while(rs.next()) {
        int id = rs.getInt(1);
        String title = rs.getString(2);
        String author = rs.getString(3);
        String genre = rs.getString(4);
        int categoryId = rs.getInt(5);

        // Do something (probably construct an object)
        // with the values here
    }
} catch(SQLException e) {
    // ERROR
} finally {
    if (rs != null) { rs.close(); }
    if (stmt != null) { stmt.close(); }
}
```

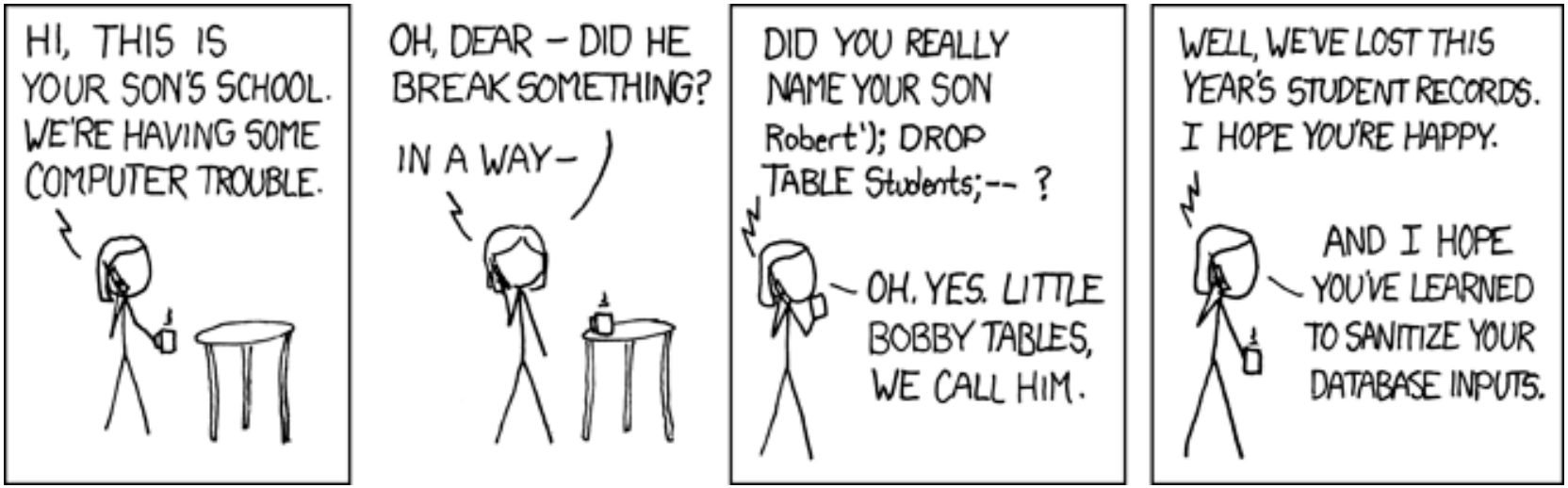
Execute an Insert, Update, or Delete

```
PreparedStatement stmt = null;
try {
    String sql = "update book " +
                 "set title = ?, author = ?, genre = ? " +
                 "where id = ?";
    stmt = connection.prepareStatement(sql);
    stmt.setString(1, book.getTitle());
    stmt.setString(2, book.getAuthor());
    stmt.setString(3, book.getGenre());
    stmt.setInt(4, book.getID());

    if(stmt.executeUpdate() == 1) {
        // OK
    } else {
        // ERROR
    }
} catch(SQLException e) {
    // ERROR
} finally {
    if (stmt != null) { stmt.close(); }
}
```

Assumes we have a reference named 'book' to an object that contains the values we need.

Prevent SQL Injection Attacks with Parameter Replacement in PreparedStatements



Commit or Rollback the Transaction & Close the Database Connection

```
try {
    …
    connection.commit();
} catch (SQLException e) {
    if(connection != null) {
        connection.rollback();
    }
} finally {
    if(connection != null) {
        connection.close();
    }
}

connection = null;
```

Putting It All Together

- Code Example
  - DatabaseAccessExample.java
  - Book.java

Retrieving Auto-increment IDs (from SQLite)

```
PreparedStatement stmt = null;
Statement keyStmt = null;
ResultSet keyRS = null;
try {
    String sql = "insert into book (title, author, genre) values (?, ?, ?)";
    stmt = connection.prepareStatement(sql);
    stmt.setString(1, book.getTitle());
    stmt.setString(2, book.getAuthor());
    stmt.setString(3, book.getGenre());

    if (stmt.executeUpdate() == 1) {
        keyStmt = connection.createStatement();
        keyRS = keyStmt.executeQuery("select last_insert_rowid()");
        keyRS.next();
        int id = keyRS.getInt(1);   // ID of the new book
        book.setID(id);
    } else {
        // ERROR
    }
} catch (SQLException e) {
    // ERROR
} finally {
    if (stmt != null) stmt.close();
    if (keyRS != null) keyRS.close();
    if (keyStmt != null) keyStmt.close();
}
```

The SQLite RDMS

SQLite

- A lightweight (simple to use) RDBMS
- Open-Source
- Simple to use and setup (compared to other RDBMSs)
- Pre-installed on recent versions of MacOS
- Pre-installed on Linux lab machines
- Easy to install on Linux and Windows

Installing on MacOS

- Do nothing. It’s already there.

Installing on Linux

- Download the source file from (usually the second file listed) http://www.sqlite.org/download.html
- tar –xzvf the downloaded file
- cd to the new folder
- ./configure
- make
- make install

Installing on Windows

- Download the first two zip files from the section labeled **Precompiled Binaries for Windows**.
- Unzip them and place the three resulting files in C:\WINDOWS\system32 (or any directory on you PATH.
  - Alternative: I created a new directory called SQLite in C:\Program Files (x86) and placed the three files in that location. I then extended the PATH variable to search that location

Installing a GUI Browser/Admin Tool

- Stand-Alone Tool (recommended)
  - Download and install DB Browser for SQLite (https://sqlitebrowser.org/)
- Browser Extensions
  - There are extensions for the popular browsers (find them with a Google search)

Accessing SQLite from Java: Manual Library Install

- Use this method if you are not using a dependency manager such as Gradle or Maven
- Download the latest JDBC driver
  - https://bitbucket.org/xerial/sqlite-jdbc/downloads/
- All IDEs
  - Create a folder called ‘lib’ in your project folder.
  - Copy the JDBC driver .jar file into the ‘lib’ folder.
- Intellij
  - Go to File -> Project Structure
  - Project Settings -> Libraries -> “+” sign -> Java
  - Select the .jar file from the ‘lib’ directory
  - Press OK until you are back at the main window
- Android Studio (preferred method is to use Gradle)
  - Go to File -> Project Structure
  - Modules -> Your Module (probably named app) -> Dependencies -> “+” sign -> Jar Dependency
  - Select the .jar file from the ‘lib’ directory
  - Press OK until you are back at the main window
- Eclipse
  - Refresh your project in eclipse.
  - Select the .jar file from the ‘lib’ directory, right click and select Build Path -> Add to Build Path

Accessing SQLite from Java: Using the Gradle Dependency Manager

- Open your project's build.gradle file
- Add the following entry (if a later version is available, update the version number):

```
dependencies {
compile group:'org.xerial', name:'sqlite-jdbc', version:'3.21.0'
}
```

Note: You probably already have a dependencies property. If so, just add the bold line to your existing entry.

Accessing SQLite from Java: Using the Maven Dependency Manager

- Open your project's pom.xml file
- Create a `<dependencies>` tag if you don't already have one
- Add the following dependency inside your dependencies tag (if a later version is available, update the version number in the version tag):

```
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.21.0</version>
</dependency>
```