**Analysis of Tables and Potential Keys:**

1. **departments**: dept\_no is the PK, inherently unique. No changes needed.
2. **titles**: title\_id is the PK, inherently unique. No changes needed.
3. **employees**: emp\_no is the PK, inherently unique. No changes needed. However, note that emp\_title\_id is a foreign key referencing titles.
4. **dept\_emp**: This is a *linking table* representing a many-to-many relationship between employees and departments. An employee can be in multiple departments, and a department can have multiple employees. The combination of emp\_no and dept\_no *should* be unique (an employee wouldn't have the same job in the same department twice simultaneously), so these columns should form a composite primary key.
5. **dept\_manager**: Similar to dept\_emp, this table likely represents a many-to-many relationship (although possibly one-to-many depending on business rules — a department could have multiple managers, and a manager *might* manage multiple departments). The combination of dept\_no and emp\_no is likely intended to be unique and would form a composite primary key.
6. **salaries**: emp\_no is likely intended to be unique at any given time (an employee should only have one salary at a given moment). If the database is designed to track salary history, then the combination of emp\_no and from\_date (not in the provided data, but typical in salary history tables) would be unique, requiring a composite key. The data provided only has emp\_no and salary, thus it may need additional considerations depending on if salary history needs to be stored. *If salary history is not required*, then emp\_no should be unique and sufficient as a primary key.

QuickDBD can generate the relationships, but **one still has to explicitly define the primary key constraint in the generated SQL.** QuickDBD's simplified syntax is for the *diagram*, not the full SQL DDL.

Therefore, one has to do the following to implement composite keys:

1. **Generate the SQL:** Use the QuickDBD code to generate the SQL.
2. **Modify the SQL (Crucial Step):** In the generated SQL, one adds the primary key constraints for the tables that require composite keys (dept\_emp and dept\_manager). There are two ways to do this:

* **Inline Constraints:** Define primary and foreign key constraints inline within the CREATE TABLE statements, keeping related constraint information together with the table definition.
* **Separate ALTER TABLE Statements:** Create the tables first *without* constraints and then add the primary and foreign key constraints later using separate ALTER TABLE statements. This style can be useful in situations where you need to create tables initially and then add constraints later due to dependencies or other factors.

The most common and generally recommended approach is the inline constraint method (Version 1). It makes the schema clearer and easier to manage because all table-related constraints are kept together.

**Explanation:**

* **NOT NULL Constraints:** Added appropriate NOT NULL constraints to columns like dept\_name, title, emp\_title\_id, etc. to the QuickDBD generated code. These are required fields and enforcing NOT NULL improves data integrity. I made an educated guess here based upon typical database design.
* **Combined and Simplified:** Removed unnecessary quoting of identifiers (double quotes) from the QuickDBD generated SQL code
* **Clarity and Consistency:** Ensured consistent formatting and removed redundancy.

**Explanation of Data Analysis Inquiries:**

* **Table Aliases:** Using aliases like e for employees, d for departments, etc., makes the queries more concise and readable.
* **JOIN Clauses:** Explicit JOIN syntax (e.g., JOIN salaries AS s ON e.emp\_no = s.emp\_no) is used for clarity and best practice.
* **WHERE Clause Filters:** Precise WHERE clauses are used to filter data according to the requirements of each query.
* **EXTRACT Function:** The EXTRACT(YEAR FROM hire\_date) function is used to correctly filter by the hire year in query #2.
* **LIKE Operator:** The LIKE operator with wildcard % is used for partial string matching in query #5.
* **IN Operator:** The IN operator is used to efficiently check for multiple values in query #7.
* **COUNT(\*) and GROUP BY:** Query #8 uses COUNT(\*) with GROUP BY to calculate frequency counts of last names.
* **ORDER BY Clause:** The ORDER BY clause sorts the results in descending order of frequency in query #8.