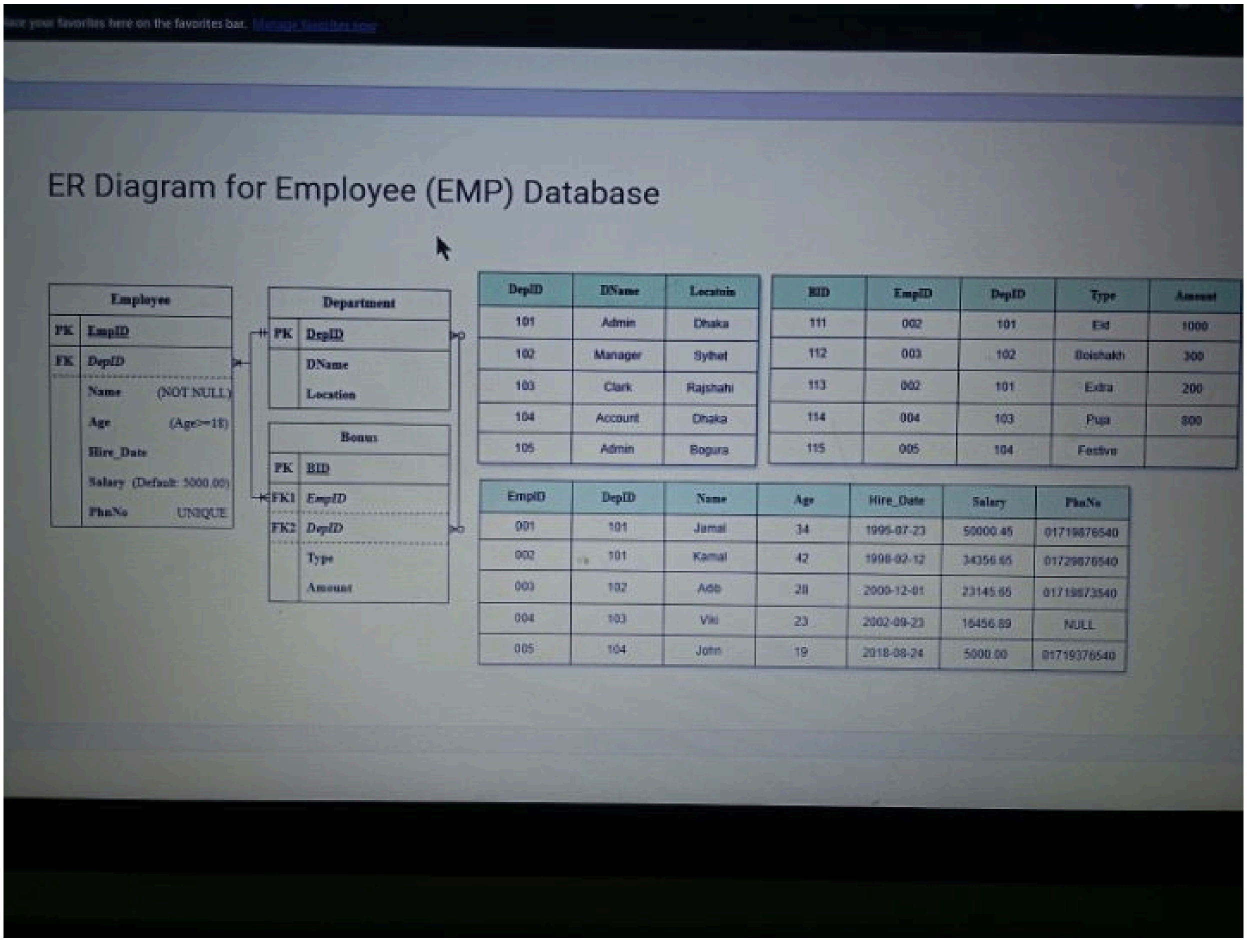
…ace your favorites here on the favorites bar. Manage favorites now

ER Diagram for Employee (EMP) Database

Employee

PK	EmpID
FK	DepID
	Name (NOT NULL)
	Age (Age>=18)
	Hire_Date
	Salary (Default: 5000.00)
	PhnNo UNIQUE

Department

PK	DepID
	DName
	Location

Bonus

PK	BID
FK1	EmpID
FK2	DepID
	Type
	Amount

DepID	DName	Locatoin
101	Admin	Dhaka
102	Manager	Sylhet
103	Clark	Rajshahi
104	Account	Dhaka
105	Admin	Bogura

BID	EmpID	DepID	Type	Amount
111	002	101	Eid	1000
112	003	102	Boishakh	300
113	002	101	Extra	200
114	004	103	Puja	800
115	005	104	Festive	

EmpID	DepID	Name	Age	Hire_Date	Salary	PhnNo
001	101	Jamal	34	1995-07-23	50000.45	01719876540
002	101	Kamal	42	1998-02-12	34356.65	01729876540
003	102	Adib	28	2000-12-01	23145.65	01719873540
004	103	Viki	23	2002-09-23	16456.89	NULL
005	104	John	19	2018-08-24	5000.00	01719376540

The image shows a slide titled "ER Diagram for Employee (EMP) Database".

On the left, it displays an Entity-Relationship Diagram with three entities: Employee, Department, and Bonus.

- **Employee** has attributes like EmpID (primary key), DepID (foreign key referencing Department), Name (not null), Age (greater than 18), Hire Date, Salary (defaulting to 5000.00), and PhnNo (unique).
- **Department** has attributes DepID (primary key), DName, and Location.
- **Bonus** has attributes BID (primary key), EmpID (foreign key referencing Employee), DepID (foreign key referencing Department), Type, and Amount.

The diagram depicts relationships between these entities.

On the right, the slide contains two tables with sample data:

- The first table, likely representing the **Department** entity, shows DepID, DName, and Location for five departments (Admin, Manager, Clark, Account, Admin) with locations (Dhaka, Sylhet, Rajshahi, Dhaka, Bogura).

- The second table, probably representing the **Employee** entity, displays EmpID, DepID, Name, Age, Hire Date, Salary, and PhnNo for five employees (Jamal, Kamal, Adib, Viki, John). It shows their respective details, including salaries and phone numbers.

The slide provides a visual representation of the database structure and sample data for an employee management system. It highlights the relationships between employees, departments, and bonuses, as well as the attributes associated with each entity.

Drop unique constraint from PhnNo in Employee Table

```
SQL
ALTER TABLE Employee
DROP CONSTRAINT IF EXISTS unique_PhnNo; -- Assuming the constraint name is unique_PhnNo

-- If you don't know the constraint name, you can use this:
-- ALTER TABLE Employee
-- DROP INDEX IF EXISTS unique_PhnNo;
```

This SQL command will remove the unique constraint from the `PhnNo` column in the `Employee` table. After executing this command, multiple employees can have the same phone number.

Important Notes:

- **Constraint Name:** If you know the name of the unique constraint on the `PhnNo` column, replace `unique_PhnNo` with the actual constraint name.
- **Database System:** The exact syntax might vary slightly depending on the database system you are using (e.g., MySQL, PostgreSQL, SQL Server, Oracle). The provided code is standard SQL and should work on most systems.
- **Impact:** Removing the unique constraint might lead to data redundancy and inconsistency. Ensure this change aligns with your application's requirements.

Please let me know if you have any other questions or need further assistance!

Show all information about admin from Department Table

```
SQL
SELECT *
FROM Department
WHERE DName = 'Admin';
```

This SQL query will:

- **SELECT ***: This selects all columns from the `Department` table.

- **FROM Department**: This specifies that the data should be retrieved from the `Department` table.
- **WHERE DName = 'Admin'**: This filters the results to only include rows where the `DName` column is equal to 'Admin'.

This query will return all the information (DepID, DName, and Location) for all departments named 'Admin' in the `Department` table.

Note: If there are multiple departments with the name 'Admin', the query will return information about all of them.

Insert values into the table mentioned above

Okay, let's insert some values into the tables based on the information in the image. I'll provide SQL `INSERT` statements. Remember, these are examples, and you might need to adjust them slightly depending on your specific database system (MySQL, PostgreSQL, SQL Server, etc.).

Department Table:

SQL
```
INSERT INTO Department (DepID, DName, Location) VALUES
(101, 'Admin', 'Dhaka'),
(102, 'Manager', 'Sylhet'),
(103, 'Clark', 'Rajshahi'),
(104, 'Account', 'Dhaka'),
(105, 'Admin', 'Bogura');
```

Employee Table:

SQL
```
INSERT INTO Employee (EmpID, DepID, Name, Age, Hire_Date, Salary, PhnNo) VALUES
(001, 101, 'Jamal', 34, '1995-07-23', 50000.45, '01719876540'),
(002, 101, 'Kamal', 42, '1998-02-12', 34356.65, '01729876540'),
(003, 102, 'Adib', 28, '2000-12-01', 23145.65, '01719873540'),
(004, 103, 'Viki', 23, '2002-09-23', 16456.89, NULL),  -- Note the NULL for PhnNo
(005, 104, 'John', 19, '2018-08-24', 5000.00, '01719376540');
```

Bonus Table:

SQL
```
INSERT INTO Bonus (BID, EmpID, DepID, Type, Amount) VALUES
(111, 002, 101, 'Eid', 1000),
(112, 003, 102, 'Boishakh', 300),
(113, 002, 101, 'Extra', 200),
(114, 004, 103, 'Puja', 800),
(115, 005, 104, 'Festive', NULL); -- Note the NULL for Amount (if allowed)
```

Key Points and Potential Adjustments:

- **Data Types:** Double-check that the data types in your database tables match the data being inserted. For example, `Hire_Date` should be a `DATE` or `DATETIME` type, `Salary` and `Amount` should be numeric (e.g., `DECIMAL` or `FLOAT`), and `PhnNo` should be a string type.
- **EmpID as Integer:** I've assumed `EmpID` is an integer. If it's a string, enclose the values in single quotes (e.g., '001').
- **Auto-Increment:** If `EmpID` or `BID` are auto-incrementing primary keys, you might not need to provide values for them in the `INSERT` statements. The database will handle it. You would just specify the columns you *are* providing values for. For example: `INSERT INTO Employee (DepID, Name, Age, Hire_Date, Salary, PhnNo) VALUES (...)`
- **NULL Values:** I've shown how to insert `NULL` values where they appear in your example data. Make sure your table definitions allow `NULL`s in those columns.
- **Database Specifics:** The exact syntax for inserting dates and handling `NULL`s might vary slightly between database systems. Consult your database documentation if you encounter errors.

After running these `INSERT` statements, your tables should be populated with the data shown in the image. Let me know if you have any other questions!

List name of all employees with their Hire_date in ascending order

SQL

```
SELECT Name, Hire_Date
FROM Employee
ORDER BY Hire_Date ASC;
```

This SQL query will:

- **SELECT Name, Hire_Date**: This selects the `Name` and `Hire_Date` columns from the `Employee` table.
- **FROM Employee**: This specifies that the data should be retrieved from the `Employee` table.
- **ORDER BY Hire_Date ASC**: This sorts the results in ascending order based on the `Hire_Date` column. `ASC` stands for ascending order (oldest hire date first).

This query will return a list of all employee names and their hire dates, sorted from the oldest hire date to the newest.

Count the number of employees with age is greater than 25

Based on the provided data in the image, here's how we can count the number of employees with an age greater than 25:

Employee Table:

EmpID	DepID	Name	Age	Hire_Date	Salary	PhnNo
001	101	Jamal	34	1995-07-23	50000.45	01719876540
002	101	Kamal	42	1998-02-12	34356.65	01729876540
003	102	Adib	28	2000-12-01	23145.65	01719873540
004	103	Viki	23	2002-09-23	16456.89	NULL
005	104	John	19	2018-08-24	5000.00	01719376540

Export to Sheets

Analysis:

- **Jamal:** Age 34 (greater than 25)
- **Kamal:** Age 42 (greater than 25)
- **Adib:** Age 28 (greater than 25)
- **Viki:** Age 23 (not greater than 25)
- **John:** Age 19 (not greater than 25)

Result:

There are **3** employees with an age greater than 25.

Find the names of employees that begin with A

SQL

```
SELECT Name
FROM Employee
WHERE Name LIKE 'A%';
```

This SQL query does the following:

- **`SELECT Name`**: Specifies that you want to retrieve the values from the `Name` column.
- **`FROM Employee`**: Indicates that you are querying the `Employee` table.
- **`WHERE Name LIKE 'A%'`**: This is the filtering condition.
    - `LIKE` is used for pattern matching.
    - `'A%'` is the pattern. `A` is the starting character you're looking for. `%` is a wildcard that matches any sequence of zero or more characters. So, `A%` matches any string that begins with the letter "A".

This query will return a list of all employee names in the `Employee` table that start with the letter "A". In your example data, it would return:

```
Name
----
Adib
```

Find all the employees whose salary is between 5000 to 10000

SQL

```
SELECT *  -- Or specify the columns you want: e.g., SELECT EmpID, Name, Salary
FROM Employee
WHERE Salary BETWEEN 5000 AND 10000;
```

Explanation:

1. **`SELECT *`**: This selects all columns from the `Employee` table. If you only need specific columns (like `EmpID`, `Name`, `Salary`), you can list them instead (e.g., `SELECT EmpID, Name, Salary`).
2. **`FROM Employee`**: This indicates that you are retrieving data from the `Employee` table.
3. **`WHERE Salary BETWEEN 5000 AND 10000`**: This is the filtering condition.
    - `BETWEEN` is an operator that checks if a value is within a specified range (inclusive).
    - `5000 AND 10000` defines the range. The query will include employees whose salary is *greater than or equal to* 5000 and *less than or equal to* 10000.

Based on your sample data:

Only John (EmpID 005) has a salary of 5000. No one has a salary between 5000 and 10000 *exclusive*. If you wanted to include salaries *up to* but not including 10000, you could use:

SQL

```
SELECT *
FROM Employee
WHERE Salary >= 5000 AND Salary < 10000;
```

Or, to find employees strictly *between* 5000 and 10000 (not including 5000 or 10000):

SQL

```
SELECT *
FROM Employee
WHERE Salary > 5000 AND Salary < 10000;
```

Change the Column Name Hire_Date to HDate in employee table

SQL

```
ALTER TABLE Employee
CHANGE COLUMN Hire_Date HDate DATE;  -- For MySQL

-- OR (depending on your database system)

ALTER TABLE Employee
RENAME COLUMN Hire_Date TO HDate; -- For some other systems like PostgreSQL
```

Explanation:

- **`ALTER TABLE Employee`**: This specifies that you are modifying the `Employee` table.
- **`CHANGE COLUMN Hire_Date HDate DATE` (MySQL)**: This is the syntax for MySQL. It renames the column `Hire_Date` to `HDate`. The `DATE` keyword (or the appropriate data type) is included to ensure that the column's data type remains consistent. If your `Hire_Date` column has a different data type (e.g., `DATETIME`), use that data type instead of `DATE`.
- **`RENAME COLUMN Hire_Date TO HDate` (Other systems)**: This is a more general SQL syntax and might work on other database systems like PostgreSQL or SQL Server.

Important Notes:

- **Database System:** The exact syntax for renaming a column can vary slightly depending on the specific database system you're using (MySQL, PostgreSQL, SQL Server, Oracle, etc.). The examples above cover the most common cases. Consult your database documentation if you encounter errors.
- **Data Type:** It's a good practice to include the data type when renaming a column (as shown in the MySQL example). This ensures that the column's data type is preserved.

- **Case Sensitivity:** Column names might be case-sensitive in some database systems. Be sure to use the correct capitalization when referring to the column.
- **Dependencies:** If other database objects (like views, stored procedures, or triggers) depend on the `Hire_Date` column, you'll need to update those objects to refer to the new `HDate` column name after you make this change.

Fetch all the data from the bonus table by adding 1000 with amount

SQL
```
SELECT BID, EmpID, DepID, Type, Amount + 1000 AS IncreasedAmount
FROM Bonus;
```

Explanation:

1. **`SELECT BID, EmpID, DepID, Type, Amount + 1000 AS IncreasedAmount`**: This selects the columns `BID`, `EmpID`, `DepID`, and `Type` from the `Bonus` table. Critically, it also calculates `Amount + 1000`. The `AS IncreasedAmount` part gives the calculated column a more descriptive name (you could choose any name you like).
2. **`FROM Bonus`**: This specifies that you are retrieving data from the `Bonus` table.

This query will return all rows from the `Bonus` table, with an additional column named `IncreasedAmount` that shows the result of adding 1000 to the original `Amount`. The original `Amount` column remains unchanged in the table itself; this query just calculates and displays the increased amount.

Show the number of Distinct location from the department table

SQL
```
SELECT COUNT(DISTINCT Location) AS NumberOfDistinctLocations
FROM Department;
```

Explanation:

1. **`SELECT COUNT(DISTINCT Location) AS NumberOfDistinctLocations`**:
    - `COUNT()` is an aggregate function that counts rows.
    - `DISTINCT Location` ensures that only unique values in the `Location` column are counted. If there are multiple departments in the same location, that location is only counted once.
    - `AS NumberOfDistinctLocations` assigns a descriptive alias to the resulting count, making it easier to understand the output.
2. **`FROM Department`**: This specifies that you are querying the `Department` table.

This query will return a single row with one column named `NumberOfDistinctLocations`, containing the total number of unique locations present in the `Department` table. Based on your sample data, the query would return 4 (Dhaka, Sylhet, Rajshahi, Bogura).

Select all employees who works in admin department

SQL

```
SELECT e.*
FROM Employee e
JOIN Department d ON e.DepID = d.DepID
WHERE d.DName = 'Admin';
```

Explanation:

1. **`SELECT e.*`**: This selects all columns from the `Employee` table (aliased as `e`). The alias makes the query more readable.
2. **`FROM Employee e`**: This specifies the `Employee` table as the primary table in the query and assigns it the alias `e`.
3. **`JOIN Department d ON e.DepID = d.DepID`**: This is a join clause that combines rows from the `Employee` and `Department` tables.
    - `JOIN Department d`: Specifies the `Department` table (aliased as `d`) to join with.
    - `ON e.DepID = d.DepID`: Specifies the join condition. It matches rows where the `DepID` column in the `Employee` table is equal to the `DepID` column in the `Department` table. This is how you link employees to their departments.
4. **`WHERE d.DName = 'Admin'`**: This filters the joined results to include only employees who work in the 'Admin' department. It checks the `DName` column in the `Department` table.

Alternative using a subquery:

SQL

```
SELECT *
FROM Employee
WHERE DepID IN (SELECT DepID FROM Department WHERE DName = 'Admin');
```

This version uses a subquery to first find the `DepID`s of the 'Admin' department and then selects employees whose `DepID` matches one of those.

Both queries achieve the same result. The `JOIN` approach is often considered more efficient, especially for larger datasets.

Find all location end with 'a' and 5 characters in length from Department table

SQL

```
SELECT Location
FROM Department
WHERE Location LIKE '____a'; -- Four underscores followed by 'a'
```

Explanation:

1. **`SELECT Location`**: This specifies that you want to retrieve the values from the `Location` column.
2. **`FROM Department`**: This indicates that you are querying the `Department` table.
3. **`WHERE Location LIKE '____a'`**: This is the filtering condition.
    - `LIKE` is used for pattern matching.
    - `'____a'` is the pattern.
        - Each underscore (`_`) matches any single character. So, four underscores match any four characters.
        - `a` matches the literal character 'a'.
        - Therefore, `'____a'` matches any string that is exactly 5 characters long and ends with the letter 'a'.

This query will return a list of all locations in the `Department` table that meet the specified criteria. Based on the sample data you provided, it would return:

```
Location
--------
Dhaka
Dhaka
```

Find the name of the employee who gets the maximum salary

SQL

```
SELECT Name
FROM Employee
ORDER BY Salary DESC
LIMIT 1;
```

Explanation:

1. **`SELECT Name`**: This specifies that you want to retrieve the `Name` of the employee.
2. **`FROM Employee`**: This indicates that you're querying the `Employee` table.

3. **`ORDER BY Salary DESC`**: This sorts the employees in descending order based on their `Salary`. The employee with the highest salary will be at the top.
4. **`LIMIT 1`**: This clause restricts the result set to only the first row, which corresponds to the employee with the highest salary since we sorted in descending order.

Alternative using a subquery (more portable across database systems):

SQL

```
SELECT Name
FROM Employee
WHERE Salary = (SELECT MAX(Salary) FROM Employee);
```

This version first uses a subquery `(SELECT MAX(Salary) FROM Employee)` to determine the maximum salary in the table. Then, it selects the `Name` of the employee(s) whose salary matches that maximum value. This form is often more efficient in some database systems.

Both versions achieve the same result. Choose the one that best suits your needs and database system.

Retrieve all information of employees along with their department name

SQL

```
SELECT e.*, d.DName AS DepartmentName
FROM Employee e
JOIN Department d ON e.DepID = d.DepID;
```

This query will retrieve all the information of employees along with their department names.

Here's a breakdown of the query:

- `SELECT e.*, d.DName AS DepartmentName`: This selects all columns (`*`) from the `Employee` table (aliased as `e`) and the `DName` column from the `Department` table (aliased as `d`). It also renames the `DName` column to `DepartmentName` using the `AS` keyword.
- `FROM Employee e`: This specifies that the data should be retrieved from the `Employee` table, aliasing it as `e`.
- `JOIN Department d ON e.DepID = d.DepID`: This joins the `Employee` and `Department` tables based on the `DepID` column, which is common to both tables.

The result will be a table with all the columns from the `Employee` table and an additional column named `DepartmentName` containing the corresponding department name for each employee.

✓ Insert values into the table mentioned above

It seems you're asking to insert values into the `Employee` and `Department` tables again. I've already provided `INSERT` statements for those tables earlier in our conversation.

To avoid redundant data, I'll provide some *new* sample data for insertion. If you want to insert the *original* data from the image, you can scroll back to find those `INSERT` statements.

New Sample Data for Department Table:

SQL

```
INSERT INTO Department (DepID, DName, Location) VALUES
(106, 'Marketing', 'Dhaka'),
(107, 'Sales', 'Chittagong'),
(108, 'HR', 'Khulna');
```

New Sample Data for Employee Table:

SQL

```
INSERT INTO Employee (EmpID, DepID, Name, Age, HDate, Salary, PhnNo) VALUES
-- Remember, we renamed Hire_Date to HDate
(006, 105, 'Anika', 28, '2020-05-15', 60000.00, '01812345678'),
(007, 106, 'Rahim', 32, '2019-08-21', 55000.50, '01798765432'),
(008, 107, 'Karim', 25, '2022-01-10', 48000.75, '01911223344'),
(009, 108, 'Sarah', 35, '2018-06-03', 70000.25, '01678901234');
```

Key Points and Reminders:

- **Auto-incrementing IDs:** If `EmpID` or `DepID` are auto-incrementing, you might need to adjust the `INSERT` statements accordingly (omitting those columns from the column list).
- **Data Types:** Ensure the data types in your table match the values being inserted.
- **Constraints:** Be mindful of any constraints (e.g., `NOT NULL`, `UNIQUE`) on the columns.
- **Database System:** The exact syntax might vary slightly depending on your database system.

If you have any specific requirements for the data you want to insert (e.g., certain ages, departments, etc.), please provide those details, and I'll tailor the `INSERT` statements accordingly.