ER → Relational Mapping Cheatsheet

1. Strong Entity → Table

Rule

- Entity name → Table name
- Attributes → Columns
- Primary Key → Underlined / marked as **PK**

Example

`Student(RollNo PK, Name, DOB)`

---

2. Weak Entity → Table

Rule

- Include **Owner's Primary Key**
- Include **Partial Key**
- Composite Primary Key = (OwnerPK, PartialKey)
- OwnerPK is also a **Foreign Key**

Example

`Employee(EmpID PK, Name)`

`Dependent(EmpID PK/FK, DependentName PK, Relationship)`

---

3. Attribute Mapping Rules

a) Simple Attribute

- Direct column in the same table

Example

`Name, DOB`

b) Composite Attribute

- Break into multiple columns

Example

```
Address → (Street, City, State, Pincode)
```

c) Multivalued Attribute

- Create a **new table**
- PK = (EntityPK, Attribute)

Example

```
Skill(EmpID PK/FK, SkillName PK)
```

d) Derived Attribute

- **Do NOT store**
- Compute when needed

Example

```
Age (derived from DOB)
```

---

4. Relationship Mapping

a) One-to-One (1:1)

Rule

- Add FK to the side with **total participation**
- OR merge tables if participation is mandatory on both sides

Example

```
Person(PersonID PK)

Passport(PassportNo PK, PersonID FK)
```

---

b) One-to-Many (1:N)

Rule

- Add FK of "1" side into "N" side table

Example

```
Department(DeptID PK)

Employee(EmpID PK, DeptID FK)
```

---

c) Many-to-Many (M:N)

Rule

- Create a **new table**
- Include PKs of both entities
- Add relationship attributes if any

Example

```
Enrolls(RollNo PK/FK, CourseID PK/FK, Grade)
```

---

5. Complete Example

ER Diagram

- Student(RollNo, Name, DOB)
- Course(CourseID, Title)
- Enrolls (M:N) with attribute Grade

Relational Model

```
Student(RollNo PK, Name, DOB)

Course(CourseID PK, Title)

Enrolls(

  RollNo PK/FK,

  CourseID PK/FK,

  Grade

)
```

---

6. Specialization / Generalization

Rule

- Superclass table + subclass tables
- Subclass PK = Superclass PK (also FK)

Example

```
Employee(EmpID PK, Name)

Teacher(EmpID PK/FK, Subject)

Clerk(EmpID PK/FK, Grade)
```

---

7. Normalization Checklist (Before Finalizing)

1NF

- Atomic values only
- No repeating groups

2NF

- No partial dependency on part of composite PK

3NF

- No transitive dependency
- Non-key attributes depend only on PK

What Interviewers Look For ✅

- Correct mapping of **strong & weak entities**
- Proper handling of **multivalued & composite attributes**
- Correct use of **primary and foreign keys**
- Clear handling of **1:1, 1:N, M:N relationships**
- Awareness of **specialization/generalization**
- **No redundancy**, properly **normalized tables**

Example ER Diagram (Simple)

Entities:

- **Student**
  - Attributes: RollNo *(PK)*, Name, DOB
- **Course**
  - Attributes: CourseID *(PK)*, Title
- **Relationship**: `Enrolls` (M:N between Student and Course)
  - Attribute: Grade

Step 1 — Identify Strong Entities

- **Student** → table with all attributes.
  `Student(RollNo PK, Name, DOB)`
- **Course** → table with all attributes.
  `Course(CourseID PK, Title)`

Step 2 — Map M:N Relationship

- M:N relationship becomes a **separate table**.
- Include **both primary keys** from participating entities as **foreign keys**.
- Add relationship attributes.

```
Enrolls(RollNo PK, CourseID PK, Grade, FOREIGN KEY (RollNo) REFERENCES Student,
FOREIGN KEY (CourseID) REFERENCES Course)
```

Final Relational Model

1. **Student**(`RollNo` PK, `Name`, `DOB`)
2. **Course**(`CourseID` PK, `Title`)
3. **Enrolls**(`RollNo` PK, `CourseID` PK, `Grade`,
   FK(`RollNo`) → Student,
   FK(`CourseID`) → Course)

Apply normalisation 1nf ,2nf ,3nf ,bcnf on table.

📌 Example Table (Unnormalized Form - UNF)

Suppose we have the following table `STUDENT`:

StudentID	StudentNae	CourseID	CourseNae	Instructor	InstructorPhone
S1	Aman	C1,C2	DBMS,OS	Prof. A,Prof.B	1111,2222
S2	Ravi	C2	OS	Prof. B	2222

🔹 Step 1: First Normal Form (1NF)

Rule: Remove repeating groups / multivalued attributes.

- **Split multivalued `CourseID, CourseName, Instructor, InstructorPhone` into separate rows.**

1NF Table:

StudentID	StudentNae	CourseID	CourseNae	Instructor	InstructorPhone
S1	Aman	C1	DBMS	Prof. A	1111
S1	Aman	C2	OS	Prof. B	2222
S2	Ravi	C2	OS	Prof. B	2222

🔹 Step 2: Second Normal Form (2NF)

Rule: No partial dependency (table must already be in 1NF).

- **Candidate key here = `(StudentID, CourseID)`.**
- **Problem: `StudentName` depends only on `StudentID`.**
- **Also, `CourseName, Instructor, InstructorPhone` depend only on `CourseID`.**

👉 **So we decompose.**

2NF Tables:

STUDENT Table:

StudentID	**StudentNae**
S1	**Aman**
S2	**Ravi**

COURSE Table:

CourseID	**CourseNae**	**Instructor**	**InstructorPhone**
C1	**DBMS**	**Prof. A**	**1111**
C2	**OS**	**Prof. B**	**2222**

ENROLLMENT Table:

StudentID	**CourseID**
S1	**C1**
S1	**C2**
S2	**C2**

🔹 Step 3: Third Normal Form (3NF)

Rule: Remove transitive dependency (non-key attribute depending on another non-key attribute).

- **In `COURSE`, `InstructorPhone` depends on `Instructor`, not on `CourseID`.**
- **So we separate INSTRUCTOR into its own table.**

3NF Tables:

STUDENT Table:

StudentID	StudentNae
S1	Aman
S2	Ravi

COURSE Table:

CourseID	CourseNae	InstructorID
C1	DBMS	I1
C2	OS	I2

INSTRUCTOR Table:

InstructorID	Instructor	InstructorPhone
I1	Prof. A	1111
I2	Prof. B	2222

ENROLLMENT Table:

StudentID	CourseID
S1	C1
S1	C2
S2	C2

🔹 Step 4: Boyce-Codd Normal Form (BCNF)

Rule: For every functional dependency X → Y, X must be a superkey.

Check:

- `StudentID → StudentName` ✅ **(StudentID is key in STUDENT).**
- `CourseID → CourseName, InstructorID` ✅ **(CourseID is key in COURSE).**
- `InstructorID → Instructor, InstructorPhone` ✅ **(InstructorID is key in INSTRUCTOR).**
- `StudentID + CourseID` **→ Enrollment** ✅ **(composite key).**

👉 **All dependencies follow BCNF.** ✅

✅ Final BCNF Schema:

1. **STUDENT(StudentID, StudentName)**
2. **COURSE(CourseID, CourseName, InstructorID)**
3. **INSTRUCTOR(InstructorID, Instructor, InstructorPhone)**
4. **ENROLLMENT(StudentID, CourseID)**

Here are the basic SQL commands for transaction control:

Start a Transaction

```
BEGIN TRANSACTION;
```

Commit a Transaction (save changes permanently)

```
COMMIT;
```

Rollback a Transaction (undo changes)

```
ROLLBACK;
```

Savepoint (set a checkpoint inside a transaction)

```
SAVEPOINT savepoint_name
```

Rollback to Savepoint

```
ROLLBACK TO savepoint_name;
```

Release Savepoint (delete a savepoint)

```
RELEASE SAVEPOINT savepoint_name;
```

Set Transaction Isolation Level

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

✅ **Example:**

sql Copy code

```sql
BEGIN TRANSACTION;

UPDATE accounts
SET balance = balance - 500
WHERE account_id = 1;

UPDATE accounts
SET balance = balance + 500
WHERE account_id = 2;

-- If everything is correct
COMMIT;

-- If something goes wrong
-- ROLLBACK;
```

🔹 Advanced Example: Money Transfer with Savepoints

Imagine we have two tables:

```sql
CREATE TABLE Accounts (
    account_id INT PRIMARY KEY,
    name VARCHAR(100),
    balance DECIMAL(12,2)
);

INSERT INTO Accounts VALUES (1, 'Aman', 5000.00);
INSERT INTO Accounts VALUES (2, 'Ravi', 3000.00);
INSERT INTO Accounts VALUES (3, 'Kiran', 2000.00);
```

Now, we'll **transfer money from one account to another**, with safeguards.

🔹 Transaction Script

Ask In Chat

```sql
BEGIN TRANSACTION;

-- Step 1: Deduct money from Sender (Aman)
UPDATE Accounts
SET balance = balance - 2000
WHERE account_id = 1;

-- Set a savepoint after deduction
SAVEPOINT after_deduction;

-- Step 2: Add money to Receiver (Ravi)
UPDATE Accounts
SET balance = balance + 2000
WHERE account_id = 2;

-- Step 3: Extra operation (e.g., give Kiran a bonus)
UPDATE Accounts
SET balance = balance + 500
WHERE account_id = 3;

-- Suppose something goes wrong here (e.g., constraint failure)
-- We can rollback only the last part
ROLLBACK TO after_deduction;

-- Step 4: Commit remaining safe operations
COMMIT;
```

Advanced Isolation Level Example

To avoid dirty reads or race conditions:

sql

Copy code

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

UPDATE Accounts
SET balance = balance - 1000
WHERE account_id = 1;

UPDATE Accounts
SET balance = balance + 1000
WHERE account_id = 2;

COMMIT;
```

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; is the strictest isolation level in SQL.

It makes sure that transactions behave as if they were executed one after another (serially), never at the same time.

Window Functions in SQL

[Window Functions in SQL - GeeksforGeeks]