

ELECTROLUX ELECTRICITY MANAGEMENT SYSTEM (EMS)

Comprehensive Database Schema Analysis & Documentation

Project: Electricity Management System (EMS)

Database Name: electricity_ems

DBMS: MySQL 8.4

ORM: Drizzle ORM

Framework: Next.js 14 + TypeScript

Semester: 5th (Fall 2025)

Total Tables: 16

Total Relationships: 24 Foreign Keys

Table of Contents

1. Executive Summary
 2. Database Overview
 3. Entity-Relationship Model
 4. Complete Table Specifications
 5. Relationships & Cardinalities
 6. Normalization Analysis (BCNF)
 7. Constraints & Integrity Rules
 8. Business Logic & Rules
 9. Drizzle ORM to SQL Mapping
 10. VIVA Preparation: Sample Queries
 11. Theoretical Concepts Fulfilled
 12. Schema Evolution & Design Decisions
-

1. Executive Summary

1.1 Project Purpose

The Electrolux EMS is a complete electricity distribution management system designed to automate billing, meter reading, customer service, and operational workflows for an electricity distribution company.

1.2 Database Statistics

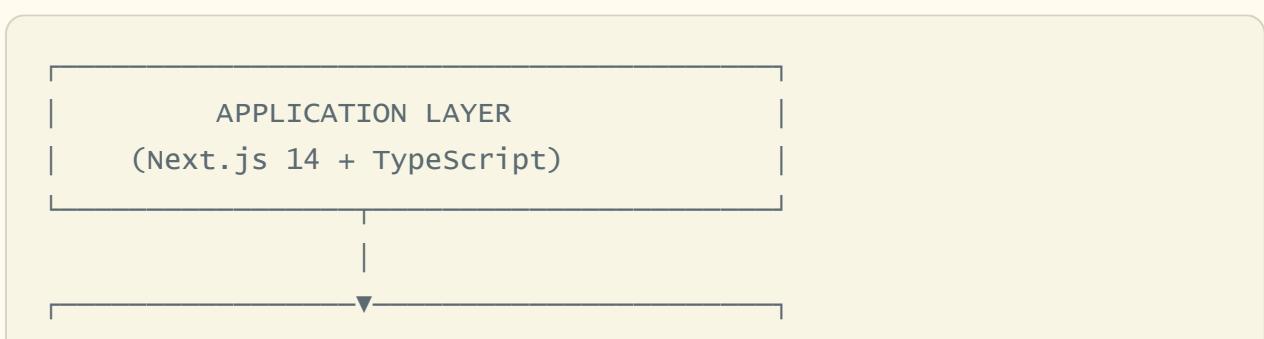
- **Total Tables:** 16
- **Total Columns:** 265+
- **Foreign Key Relationships:** 24
- **Unique Constraints:** 18
- **Enum Types:** 35+
- **Indexes:** 12
- **Normalization Level:** BCNF (Boyce-Codd Normal Form)

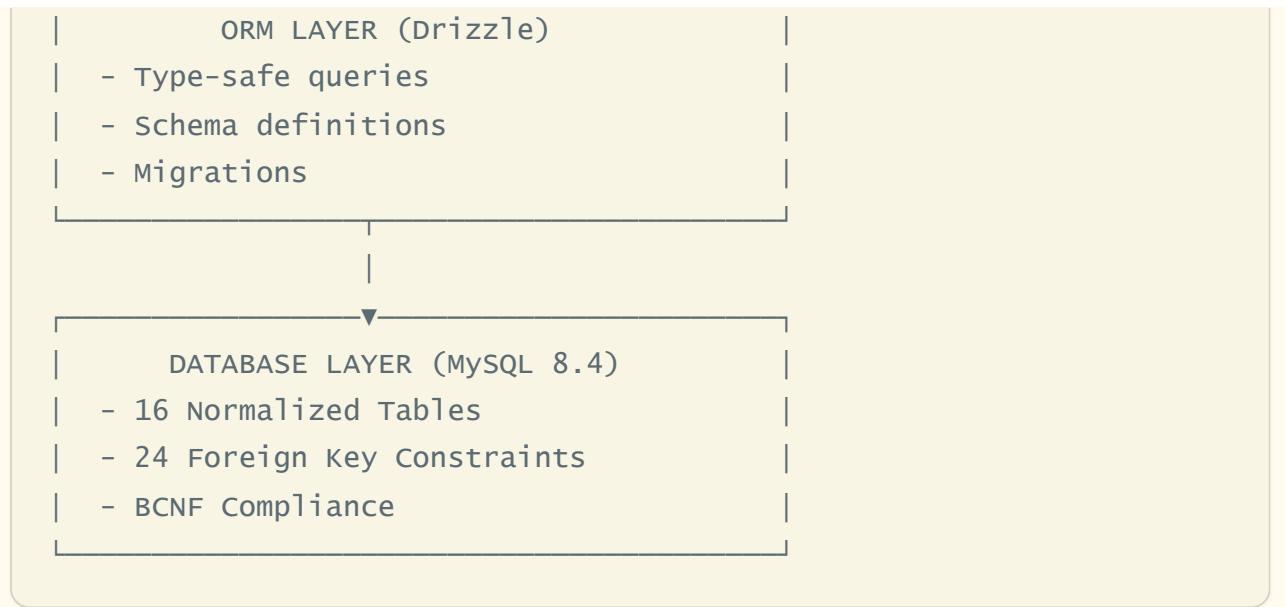
1.3 Key Features

- Multi-user system (Admin, Employee, Customer)
 - Automated billing with slab-based tariff calculation
 - Real-time meter reading workflow
 - Payment processing and tracking
 - Work order and complaint management
 - Power outage tracking and notifications
 - Connection request processing
-

2. Database Overview

2.1 Database Architecture





2.2 Table Categories

Core Authentication & User Management

1. `users` - User accounts (admin/employee/customer)
2. `customers` - Customer information and profiles
3. `employees` - Employee details and assignments
4. `password_reset_requests` - Password reset workflow

Billing & Financial

5. `bills` - Monthly electricity bills
6. `tariffs` - Pricing structure per connection type
7. `tariff_slabs` - Tiered pricing (BCNF normalized from tariffs)
8. `payments` - Payment transactions and receipts

Operational

9. `meter_readings` - Meter reading records
10. `work_orders` - Task assignments for employees
11. `complaints` - Customer complaint tracking
12. `outages` - Power outage management

Workflow & Requests

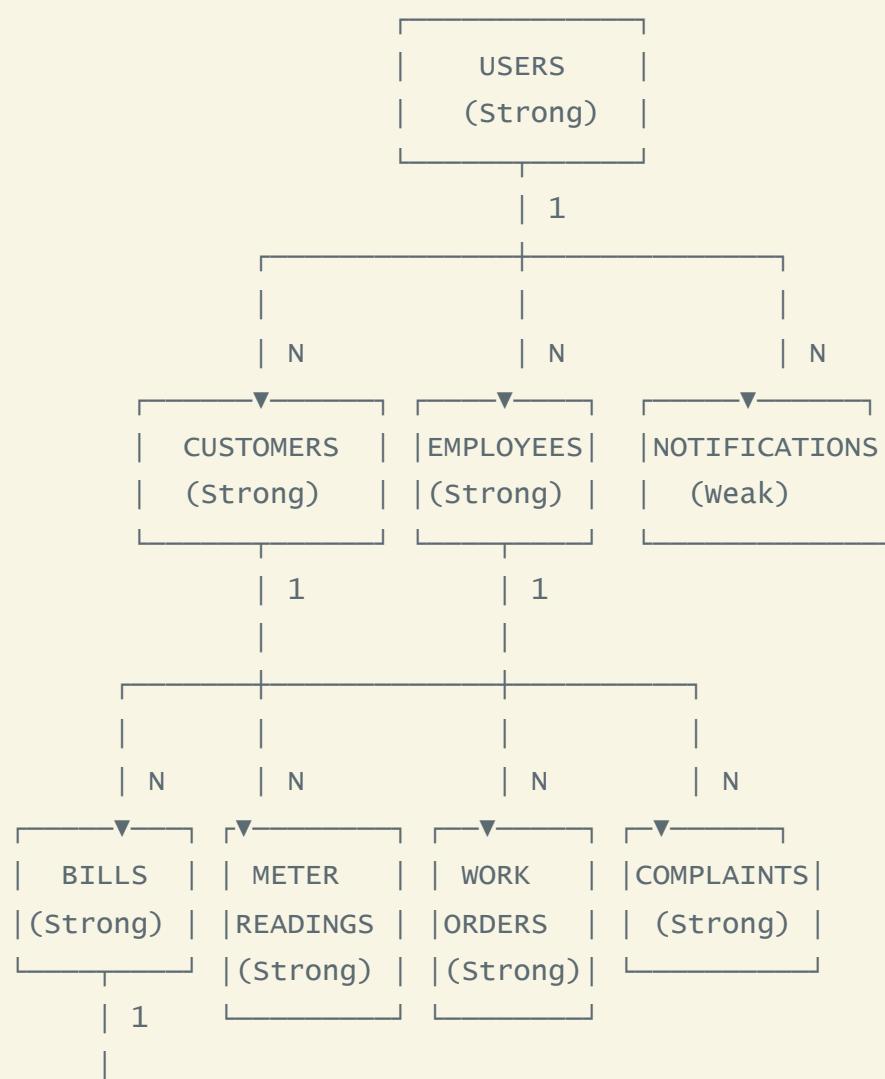
13. `connection_requests` - New connection applications
 14. `bill_requests` - Customer bill generation requests
 15. `reading_requests` - Meter reading requests

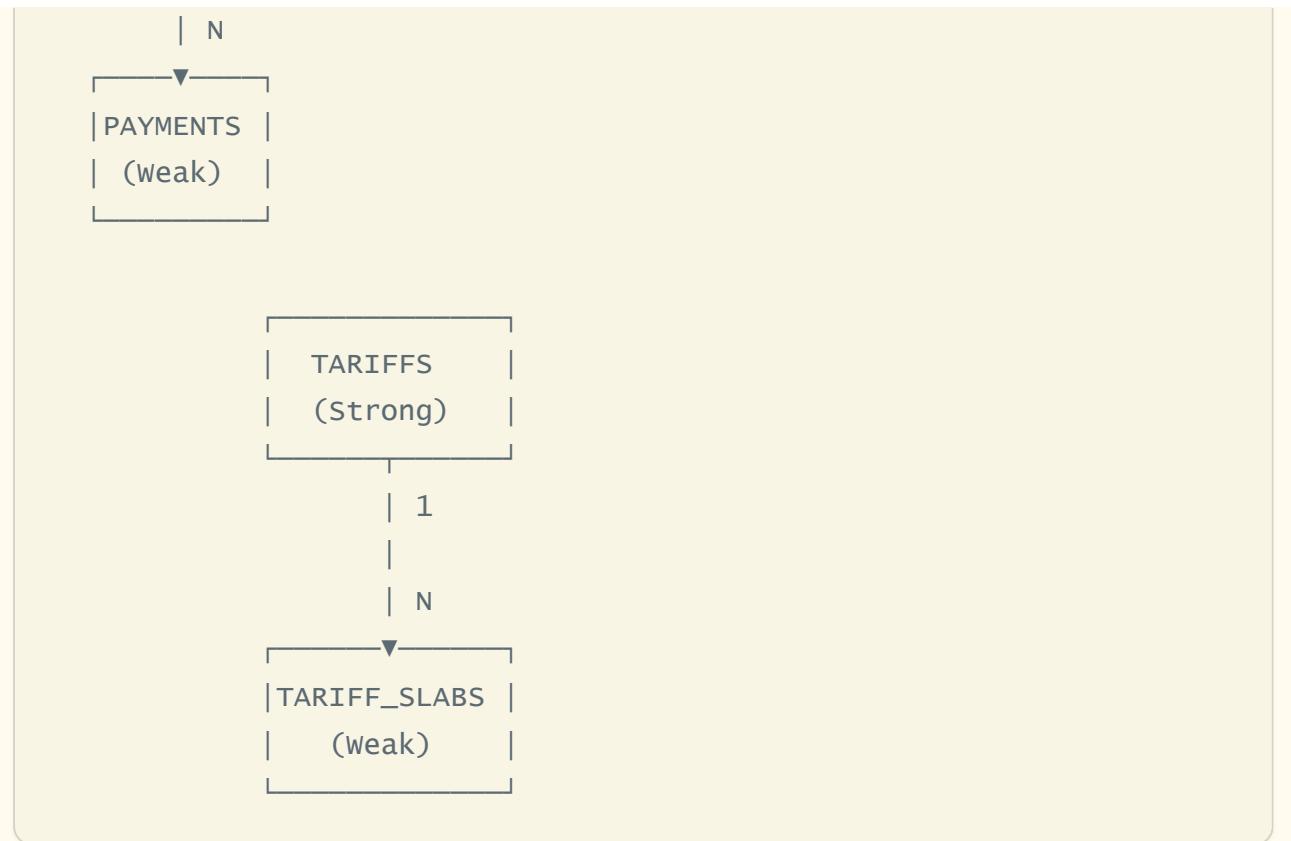
Communication

- ## 16. notifications - System notifications for users

3. Entity-Relationship Model

3.1 ER Diagram





3.2 Entity Types

Strong Entities (Independent Existence)

- `users` - Central authentication entity
- `customers` - Customer master data
- `employees` - Employee master data
- `tariffs` - Pricing structure
- `bills` - Billing records
- `meter_readings` - Reading records
- `work_orders` - Task management
- `complaints` - Complaint records
- `connection_requests` - Application records
- `outages` - Outage tracking

Weak Entities (Dependent on Strong Entities)

- `tariff_slabs` - Cannot exist without `tariffs`
- `payments` - Depends on `bills` (though has partial key)
- `notifications` - Depends on `users`
- `bill_requests` - Depends on `customers`

- `reading_requests` - Depends on `customers`
- `password_reset_requests` - Depends on `users`

3.3 Relationship Types

Relationship	Type	Cardinality	Description
Users → Customers	1:N	One-to-Many	One user account per customer
Users → Employees	1:N	One-to-Many	One user account per employee
Users → Notifications	1:N	One-to-Many	User receives many notifications
Customers → Bills	1:N	One-to-Many	Customer has many bills
Customers → Meter Readings	1:N	One-to-Many	Customer has many readings
Customers → Payments	1:N	One-to-Many	Customer makes many payments
Customers → Complaints	1:N	One-to-Many	Customer files many complaints
Employees → Meter Readings	1:N	One-to-Many	Employee records many readings
Employees → Work Orders	1:N	One-to-Many	Employee handles many tasks
Tariffs → Tariff Slabs	1:N	One-to-Many	Tariff has multiple price slabs
Bills → Payments	1:N	One-to-Many	Bill can have multiple payments
Meter Readings → Bills	1:1	One-to-One (Optional)	Reading generates one bill

4. Complete Table Specifications

4.1 USERS Table

Purpose: Central authentication and user management

Type: Strong Entity

Primary Key: `id` (AUTO_INCREMENT)

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Unique identifier
<code>email</code>	VARCHAR(255)	NOT NULL, UNIQUE	-	Login email
<code>password</code>	VARCHAR(255)	NOT NULL	-	Hashed password (bcrypt)
<code>user_type</code>	ENUM('admin', 'employee', 'customer')	NOT NULL	-	User role
<code>name</code>	VARCHAR(255)	NOT NULL	-	Full name
<code>phone</code>	VARCHAR(20)	NULL	NULL	Contact number
<code>is_active</code>	INT	NOT NULL	1	Account status (1=active)
<code>requires_password_change</code>	INT	NOT NULL	0	Force password change flag
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	Record creation time
<code>updated_at</code>	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	Last update time

Business Rules:

- Email must be unique across all user types
- Password must be hashed using bcrypt (min 10 rounds)
- Default password for new users: `password123`

- `user_type` determines access control and permissions

Sample Data:

```
INSERT INTO users VALUES (1, 'admin@electrolux.com', '$2a$10$...',
'admin', 'Admin User', '+1234567890', 1, 0, NOW(), NOW());
```

4.2 CUSTOMERS Table

Purpose: Store customer information and account details

Type: Strong Entity (Dependent on Users)

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys: `user_id` → `users.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Customer ID
<code>user_id</code>	INT	NOT NULL, FK, UNIQUE	-	Links to users table
<code>account_number</code>	VARCHAR(50)	NOT NULL, UNIQUE	-	Account number (ELX-YYYY-XXXXXX)
<code>meter_number</code>	VARCHAR(50)	UNIQUE	NULL	Meter serial number
<code>full_name</code>	VARCHAR(255)	NOT NULL	-	Customer full name
<code>email</code>	VARCHAR(255)	NOT NULL	-	Contact email
<code>phone</code>	VARCHAR(20)	NOT NULL	-	Contact number
<code>address</code>	VARCHAR(500)	NOT NULL	-	Installation address
<code>city</code>	VARCHAR(100)	NOT NULL	-	City name
<code>state</code>	VARCHAR(100)	NOT NULL	-	State/Province
<code>pincode</code>	VARCHAR(10)	NOT NULL	-	Postal code
<code>zone</code>	VARCHAR(50)	NULL	NULL	Load shedding zone
<code>connection_type</code>	ENUM	NOT NULL	-	Residential/Commercial/Industrial/Agricultural
<code>status</code>	ENUM	NOT NULL	'active'	pending_installation/active/suspended/inactive
<code>connection_date</code>	DATE	NOT NULL	-	Connection activation date
<code>date_of_birth</code>	DATE	NULL	NULL	Customer DOB (KYC)
<code>installation_charges</code>	DECIMAL(10,2)	NULL	NULL	One-time installation fee
<code>last_bill_amount</code>	DECIMAL(10,2)	NOT NULL	0.00	Last bill total (cached)
<code>last_payment_date</code>	DATE	NULL	NULL	Last payment date (cached)
<code>average_monthly_usage</code>	DECIMAL(10,2)	NOT NULL	0.00	Avg consumption (cached)
<code>outstanding_balance</code>	DECIMAL(10,2)	NOT NULL	0.00	Denormalized - sum of unpaid bills

Column	Data Type	Constraints	Default	Description
payment_status	ENUM('paid', 'pending', 'overdue')	NOT NULL	'paid'	Current payment status
created_at	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
updated_at	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Foreign Keys:

```
CONSTRAINT fk_customer_user FOREIGN KEY (user_id) REFERENCES
users(id) ON DELETE CASCADE
```

Business Rules:

- Account number format: ELX-YYYY-XXXXXX (e.g., ELX-2024-000001)
- Meter number format: MTR-{CITY_CODE}-XXXXXX (e.g., MTR-KHI-000001)
- `outstanding_balance` is **strategically denormalized** for performance
- Status transitions: pending_installation → active → suspended/inactive
- `connection_type` determines applicable tariff

Strategic Denormalization:

- `outstanding_balance` = SUM(unpaid bills) - cached for dashboard performance
- Updated via triggers or application logic when bills/payments are created

4.3 EMPLOYEES Table

Purpose: Store employee information and assignments

Type: Strong Entity (Dependent on Users)

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys: `user_id` → `users.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Employee ID
<code>employee_number</code>	VARCHAR(20)	UNIQUE	NULL	Employee number (EMP-XXX)

Column	Data Type	Constraints	Default	Description
user_id	INT	NOT NULL, FK, UNIQUE	-	Links to users table
employee_name	VARCHAR(255)	NOT NULL	-	Full name
email	VARCHAR(255)	NOT NULL	-	Work email
phone	VARCHAR(20)	NOT NULL	-	Contact number
designation	VARCHAR(100)	NOT NULL	-	Job title (e.g., Meter Reader)
department	VARCHAR(100)	NOT NULL	-	Department name
assigned_zone	VARCHAR(100)	NULL	NULL	Work zone assignment
status	ENUM('active', 'inactive')	NOT NULL	'active'	Employment status
hire_date	DATE	NOT NULL	-	Date of joining
created_at	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
updated_at	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Business Rules:

- Employee number format: `EMP-XXX` (e.g., `EMP-001`)
- Common designations: Meter Reader, Supervisor, Technician, Field Officer
- Common departments: Operations, Billing, Maintenance, Customer Service, Technical
- Zone assignment enables geographic work distribution

4.4 BILLS Table

Purpose: Store monthly electricity bills

Type: Strong Entity

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys:

- `customer_id` → `customers.id`
- `meter_reading_id` → `meter_readings.id`
- `tariff_id` → `tariffs.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Bill ID
<code>customer_id</code>	INT	NOT NULL, FK	-	Customer reference
<code>bill_number</code>	VARCHAR(50)	NOT NULL, UNIQUE	-	Bill number (BILL-YYYY-MM-XXXX)
<code>billing_month</code>	DATE	NOT NULL	-	Billing period (YYYY-MM-01)
<code>issue_date</code>	DATE	NOT NULL	-	Bill generation date
<code>due_date</code>	DATE	NOT NULL	-	Payment deadline
<code>units_consumed</code>	DECIMAL(10,2)	NOT NULL	-	kWh consumed
<code>meter_reading_id</code>	INT	FK	NULL	Related meter reading
<code>base_amount</code>	DECIMAL(10,2)	NOT NULL	-	Energy charges (slab calculation)
<code>fixed_charges</code>	DECIMAL(10,2)	NOT NULL	-	Fixed monthly fee
<code>electricity_duty</code>	DECIMAL(10,2)	NOT NULL	0.00	Duty on base amount
<code>gst_amount</code>	DECIMAL(10,2)	NOT NULL	0.00	GST (18% on total)
<code>total_amount</code>	DECIMAL(10,2)	NOT NULL	-	Final bill amount
<code>status</code>	ENUM	NOT NULL	'generated'	generated/issued/paid/overdue/cancelled
<code>payment_date</code>	DATE	NULL	NULL	Date of payment
<code>tariff_id</code>	INT	FK	NULL	Applied tariff reference
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
<code>updated_at</code>	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Bill Calculation Formula (Whole Numbers Only):

1. $\text{baseAmount} = \sum(\text{slab_units} \times \text{rate_per_unit})$ for all applicable slabs
2. $\text{baseAmount} = \text{Math.round}(\text{baseAmount})$
3. $\text{fixedCharges} = \text{Math.round}(\text{tariff.fixedCharge})$
4. $\text{electricityDuty} = \text{Math.round}(\text{baseAmount} \times \text{duty_percent} / 100)$
5. $\text{gstAmount} = \text{Math.round}((\text{baseAmount} + \text{fixedCharges} + \text{electricityDuty}) \times 18 / 100)$
6. $\text{totalAmount} = \text{Math.round}(\text{baseAmount} + \text{fixedCharges} + \text{electricityDuty} + \text{gstAmount})$

Business Rules:

- Bill number format: BILL-YYYY-MM-XXXX (e.g., BILL-2024-11-0001)
- All monetary values are **whole numbers** (no decimal paisa)
- Duty applies to `base_amount` only (NOT fixed charges)
- GST applies to `(base_amount + fixed_charges + electricity_duty)`
- Due date = issue_date + 15 days
- Status transitions: generated → issued → paid/overdue

4.5 METER_READINGS Table

Purpose: Record meter reading history

Type: Strong Entity

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys:

- `customer_id` → `customers.id`
- `employee_id` → `employees.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Reading ID
<code>customer_id</code>	INT	NOT NULL, FK	-	Customer reference
<code>meter_number</code>	VARCHAR(50)	NOT NULL	-	Meter serial number
<code>current_reading</code>	DECIMAL(10,2)	NOT NULL	-	Current meter value (kWh)
<code>previous_reading</code>	DECIMAL(10,2)	NOT NULL	-	Previous meter value (kWh)
<code>units_consumed</code>	DECIMAL(10,2)	NOT NULL	-	Consumption (current - previous)
<code>reading_date</code>	DATE	NOT NULL	-	Reading date
<code>reading_time</code>	TIMESTAMP	NOT NULL	-	Exact reading timestamp

Column	Data Type	Constraints	Default	Description
<code>meter_condition</code>	ENUM	NOT NULL	'good'	good/fair/poor/damaged
<code>accessibility</code>	ENUM	NOT NULL	'accessible'	accessible/partially_accessible/inaccessible
<code>employee_id</code>	INT	FK	NULL	Employee who recorded
<code>photo_path</code>	VARCHAR(500)	NULL	NULL	Meter photo URL
<code>notes</code>	TEXT	NULL	NULL	Additional remarks
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
<code>updated_at</code>	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Business Rules:

- `current_reading` must be \geq `previous_reading` (monotonic)
- `units_consumed` = `current_reading` - `previous_reading`
- Photo upload is optional but recommended for disputes
- First reading for a new customer: `previous_reading` = 0 or inherited

4.6 TARIFFS Table

Purpose: Define pricing structure per connection type

Type: Strong Entity

Primary Key: `id` (AUTO_INCREMENT)

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Tariff ID
<code>category</code>	ENUM	NOT NULL	-	Residential/Commercial/Industrial/Agricultural
<code>fixed_charge</code>	DECIMAL(10,2)	NOT NULL	-	Monthly fixed fee
<code>time_of_use_peak_rate</code>	DECIMAL(10,2)	NULL	NULL	Peak hour rate (future use)
<code>time_of_use_normal_rate</code>	DECIMAL(10,2)	NULL	NULL	Normal hour rate
<code>time_of_use_offpeak_rate</code>	DECIMAL(10,2)	NULL	NULL	Off-peak hour rate
<code>electricity_duty_percent</code>	DECIMAL(5,2)	NOT NULL	0.00	Duty % on base amount
<code>gst_percent</code>	DECIMAL(5,2)	NOT NULL	18.00	GST % (default 18%)
<code>effective_date</code>	DATE	NOT NULL	-	Tariff start date
<code>valid_until</code>	DATE	NULL	NULL	Tariff end date (NULL = active)
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
<code>updated_at</code>	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Business Rules:

- One active tariff per category at any time
 - Slab rates stored in separate `tariff_slabs` table (BCNF normalization)
 - Historical tariffs preserved with `valid_until` date
-

4.7 TARIFF_SLABS Table

Purpose: Store tiered pricing for each tariff (BCNF Normalization)

Type: Weak Entity (Depends on Tariffs)

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys: `tariff_id` → `tariffs.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Slab ID
<code>tariff_id</code>	INT	NOT NULL, FK	-	Parent tariff
<code>slab_order</code>	INT	NOT NULL	-	Sequence number (1, 2, 3...)
<code>start_units</code>	INT	NOT NULL	-	Lower boundary (inclusive)
<code>end_units</code>	INT	NULL	NUL	Upper boundary (inclusive, NULL = ∞)
<code>rate_per_unit</code>	DECIMAL(10,2)	NOT NULL	-	Rate per kWh in this slab
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-

Example: Residential Tariff Slabs:

```

INSERT INTO tariff_slabs VALUES
(1, 1, 1, 0, 100, 4.50),      -- 0-100 kwh: Rs. 4.50/unit
(2, 1, 2, 101, 200, 6.00),    -- 101-200 kwh: Rs. 6.00/unit
(3, 1, 3, 201, 300, 7.50),    -- 201-300 kwh: Rs. 7.50/unit
(4, 1, 4, 301, 500, 9.00),    -- 301-500 kwh: Rs. 9.00/unit
(5, 1, 5, 501, NULL, 10.50);  -- 501+ kwh: Rs. 10.50/unit

```

Slab Calculation Algorithm:

```

let baseAmount = 0;
let remaining = unitsConsumed;

for (const slab of tariffSlabs) {
  if (remaining <= 0) break;

  const slabSpan = slab.end_units === null
    ? remaining
    : (slab.end_units - slab.start_units + 1); // Inclusive

  const unitsInSlab = Math.min(remaining, slabSpan);
  baseAmount += unitsInSlab * slab.rate_per_unit;
  remaining -= unitsInSlab;
}

baseAmount = Math.round(baseAmount); // whole number

```

4.8 PAYMENTS Table

Purpose: Track payment transactions

Type: Weak Entity (Depends on Bills)

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys:

- `customer_id` → `customers.id`
- `bill_id` → `bills.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Payment ID
<code>customer_id</code>	INT	NOT NULL, FK	-	Customer reference
<code>bill_id</code>	INT	FK	NULL	Related bill (NULL = advance)

Column	Data Type	Constraints	Default	Description
payment_amount	DECIMAL(10,2)	NOT NULL	-	Amount paid
payment_method	ENUM	NOT NULL	-	credit_card/debit_card/bank_transfer/cash/cheque/upi/wallet
payment_date	DATE	NOT NULL	-	Transaction date
transaction_id	VARCHAR(100)	UNIQUE	NULL	Bank transaction ID
receipt_number	VARCHAR(50)	UNIQUE	NULL	Receipt number (RCP-YYYY-XXXXXX)
status	ENUM	NOT NULL	'completed'	pending/completed/failed/refunded
notes	TEXT	NULL	NULL	Additional remarks
created_at	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
updated_at	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Business Rules:

- Receipt number format: RCP-YYYY-XXXXXX (e.g., RCP-2024-000001)
- Successful payment triggers bill status update to 'paid'
- Partial payments allowed (multiple payments per bill)
- Advance payments stored with bill_id = NULL

4.9 WORK_ORDERS Table

Purpose: Manage employee task assignments

Type: Strong Entity

Primary Key: id (AUTO_INCREMENT)

Foreign Keys:

- employee_id → employees.id
- customer_id → customers.id

Column	Data Type	Constraints	Default	Description
id	INT	PRIMARY KEY, AUTO_INCREMENT	-	Work order ID
employee_id	INT	FK	NULL	Assigned employee
customer_id	INT	FK	NULL	Related customer
work_type	ENUM	NOT NULL	-	meter_reading/maintenance/complaint_resolution/new_connection/disconnection/reconnection
title	VARCHAR(255)	NOT NULL	-	Task title
description	TEXT	NULL	NULL	Task details
status	ENUM	NOT NULL	'assigned'	assigned/in_progress/completed/cancelled
priority	ENUM	NOT NULL	'medium'	low/medium/high/urgent
assigned_date	DATE	NOT NULL	-	Task assignment date
due_date	DATE	NOT NULL	-	Deadline
completion_date	DATE	NULL	NULL	Actual completion date
completion_notes	TEXT	NULL	NULL	Completion remarks

Column	Data Type	Constraints	Default	Description
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
<code>updated_at</code>	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Business Rules:

- Tasks sorted by priority and due_date for employee queue
- Completion updates meter reading or complaint status
- Task history preserved for audit trail

4.10 COMPLAINTS Table

Purpose: Customer complaint management

Type: Strong Entity

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys:

- `customer_id` → `customers.id`
- `employee_id` → `employees.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Complaint ID
<code>customer_id</code>	INT	NOT NULL, FK	-	Customer reference
<code>employee_id</code>	INT	FK	NULL	Assigned employee
<code>work_order_id</code>	INT	NULL	NULL	Related work order (implicit FK)
<code>category</code>	ENUM	NOT NULL	-	power_outage/billing/service/meter_issue/connection/other
<code>title</code>	VARCHAR(255)	NOT NULL	-	Complaint title
<code>description</code>	TEXT	NOT NULL	-	Detailed description
<code>status</code>	ENUM	NOT NULL	'submitted'	submitted/under_review/assigned/in_progress/resolved/closed
<code>priority</code>	ENUM	NOT NULL	'medium'	low/medium/high/urgent
<code>resolution_notes</code>	TEXT	NULL	NULL	Resolution details
<code>submitted_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	Submission time
<code>reviewed_at</code>	TIMESTAMP	NULL	NULL	Review timestamp
<code>assigned_at</code>	TIMESTAMP	NULL	NULL	Assignment timestamp
<code>resolved_at</code>	TIMESTAMP	NULL	NULL	Resolution timestamp
<code>closed_at</code>	TIMESTAMP	NULL	NULL	Closure timestamp
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
<code>updated_at</code>	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Business Rules:

- Status lifecycle timestamps track SLA compliance
 - Creating work order auto-updates complaint status
 - Customer notified at each status change
-

4.11 NOTIFICATIONS Table

Purpose: User notifications and alerts

Type: Weak Entity (Depends on Users)

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys: `user_id` → `users.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Notification ID
<code>user_id</code>	INT	NOT NULL, FK	-	Recipient user
<code>notification_type</code>	ENUM	NOT NULL	-	billing/payment/usage/outage/service/reminder/system/maintenance/alert/info/work_order
<code>title</code>	VARCHAR(255)	NOT NULL	-	Notification title
<code>message</code>	TEXT	NOT NULL	-	Notification content
<code>priority</code>	ENUM	NOT NULL	'normal'	low/normal/medium/high
<code>action_url</code>	VARCHAR(255)	NULL	NULL	Click action URL
<code>action_text</code>	VARCHAR(100)	NULL	NULL	Action button text
<code>is_read</code>	INT	NOT NULL	0	Read status (0=unread, 1=read)
<code>read_at</code>	TIMESTAMP	NULL	NULL	Read timestamp
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-

Business Rules:

- Real-time notifications on bill generation, payment success, outage alerts
 - Unread count displayed in header
 - Auto-marked as read when action URL is clicked
-

4.12 BILL_REQUESTS Table

Purpose: Customer bill generation requests

Type: Weak Entity (Depends on Customers)

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys:

- `customer_id` → `customers.id`
- `created_by` → `users.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Request ID
<code>request_id</code>	VARCHAR(50)	NOT NULL, UNIQUE	-	Request number (BREQ-YYYY-XXXXXX)
<code>customer_id</code>	INT	NOT NULL, FK	-	Customer reference
<code>billing_month</code>	DATE	NOT NULL	-	Requested billing period
<code>priority</code>	ENUM	NOT NULL	'medium'	low/medium/high
<code>notes</code>	TEXT	NULL	NULL	Customer remarks
<code>status</code>	ENUM	NOT NULL	'pending'	pending/processing/completed/rejected
<code>request_date</code>	DATE	NOT NULL	-	Request date
<code>created_by</code>	INT	FK	NULL	User who created (NULL = customer)
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
<code>updated_at</code>	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Unique Index:

```
UNIQUE INDEX unique_request_month (customer_id, billing_month)
```

Business Rules:

- One bill request per customer per month
- Admin bulk generation processes pending requests
- Status updated to 'completed' when bill is generated

4.13 CONNECTION_REQUESTS Table

Purpose: New electricity connection applications

Type: Strong Entity

Primary Key: `id` (AUTO_INCREMENT)

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Application ID
<code>application_number</code>	VARCHAR(50)	NOT NULL, UNIQUE	-	Application number (CONN-YYYY-XXXXXX)
<code>applicant_name</code>	VARCHAR(255)	NOT NULL	-	Applicant full name
<code>father_name</code>	VARCHAR(255)	NULL	NULL	Father's name (KYC)

Column	Data Type	Constraints	Default	Description
email	VARCHAR(255)	NOT NULL	-	Contact email
phone	VARCHAR(20)	NOT NULL	-	Primary contact
alternate_phone	VARCHAR(20)	NULL	NULL	Secondary contact
id_type	ENUM	NOT NULL	-	passport/drivers_license/national_id/voter_id/aadhaar
id_number	VARCHAR(100)	NOT NULL	-	ID document number
property_type	ENUM	NOT NULL	-	Residential/Commercial/Industrial/Agricultural
connection_type	ENUM	NOT NULL	-	single-phase/three-phase/industrial
load_required	DECIMAL(10,2)	NULL	NULL	Required load (kW)
property_address	VARCHAR(500)	NOT NULL	-	Installation address
city	VARCHAR(100)	NOT NULL	-	City
state	VARCHAR(100)	NULL	NULL	State
pincode	VARCHAR(10)	NULL	NULL	Postal code
landmark	VARCHAR(255)	NULL	NULL	Nearby landmark
zone	VARCHAR(50)	NULL	NULL	Load shedding zone
preferred_date	DATE	NULL	NULL	Preferred installation date
purpose_of_connection	ENUM	NOT NULL	-	domestic/business/industrial/agricultural
existing_connection	BOOLEAN	NOT NULL	FALSE	Has existing connection?
existing_account_number	VARCHAR(50)	NULL	NULL	If transferring
status	ENUM	NOT NULL	'pending'	pending/under_review/approved/rejected/connected
estimated_charges	DECIMAL(10,2)	NULL	NULL	Installation cost estimate
inspection_date	DATE	NULL	NULL	Site inspection date
approval_date	DATE	NULL	NULL	Approval date
installation_date	DATE	NULL	NULL	Actual installation date
application_date	DATE	NOT NULL	-	Application submission date
account_number	VARCHAR(50)	NULL	NULL	Generated account number
temporary_password	VARCHAR(255)	NULL	NULL	Auto-generated password
created_at	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
updated_at	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Indexes:

```
INDEX idx_status (status);
INDEX idx_email (email);
```

Workflow:

1. **Pending** → Customer submits online application
2. **Under Review** → Admin reviews documents

3. **Approved** → Site inspection scheduled
 4. **Connected** → User + customer accounts created
 5. **Rejected** → Invalid documents or area not serviceable
-

4.14 READING_REQUESTS Table

Purpose: Customer meter reading requests

Type: Weak Entity (Depends on Customers)

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys:

- `customer_id` → `customers.id`
- `work_order_id` → `work_orders.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Request ID
<code>request_number</code>	VARCHAR(50)	NOT NULL, UNIQUE	-	Request number (RREQ-YYYY-XXXXXX)
<code>customer_id</code>	INT	NOT NULL, FK	-	Customer reference
<code>request_date</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	Request timestamp
<code>preferred_date</code>	DATE	NULL	NULL	Preferred reading date
<code>request_reason</code>	TEXT	NULL	NULL	Reason for request
<code>priority</code>	ENUM	NOT NULL	'normal'	normal/urgent
<code>status</code>	ENUM	NOT NULL	'pending'	pending/assigned/completed/cancelled
<code>notes</code>	TEXT	NULL	NULL	Additional notes
<code>work_order_id</code>	INT	FK	NULL	Generated work order
<code>assigned_date</code>	TIMESTAMP	NULL	NULL	Assignment timestamp
<code>completed_date</code>	TIMESTAMP	NULL	NULL	Completion timestamp
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
<code>updated_at</code>	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Indexes:

```

INDEX idx_status (status);
INDEX idx_customer_id (customer_id);
INDEX idx_request_date (request_date);

```

Business Rules:

- Customer can request urgent reading for dispute resolution
- System auto-creates work order when assigned to employee

4.15 OUTAGES Table

Purpose: Power outage tracking and notifications

Type: Strong Entity

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys: `created_by` → `users.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Outage ID
<code>area_name</code>	VARCHAR(255)	NOT NULL	-	Affected area name
<code>zone</code>	VARCHAR(50)	NOT NULL	-	Zone identifier
<code>outage_type</code>	ENUM('planned', 'unplanned')	NOT NULL	-	Type of outage
<code>reason</code>	TEXT	NULL	NULL	Outage reason/description
<code>severity</code>	ENUM	NOT NULL	-	low/medium/high/critical
<code>scheduled_start_time</code>	DATETIME	NULL	NULL	Planned start (for scheduled)
<code>scheduled_end_time</code>	DATETIME	NULL	NULL	Planned end (for scheduled)
<code>actual_start_time</code>	DATETIME	NULL	NULL	Actual start time
<code>actual_end_time</code>	DATETIME	NULL	NULL	Actual restoration time
<code>affected_customer_count</code>	INT	NOT NULL	0	Number of affected customers
<code>status</code>	ENUM	NOT NULL	-	scheduled/ongoing/restored/cancelled
<code>restoration_notes</code>	TEXT	NULL	NULL	Restoration details
<code>created_by</code>	INT	NOT NULL, FK	-	Admin user who created
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
<code>updated_at</code>	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Business Rules:

- Planned outages scheduled 24-48 hours in advance

- All customers in affected zone receive notifications
- Status updates trigger automatic notifications

4.16 PASSWORD_RESET_REQUESTS Table

Purpose: Password reset workflow management

Type: Weak Entity (Depends on Users)

Primary Key: `id` (AUTO_INCREMENT)

Foreign Keys:

- `user_id` → `users.id`
- `processed_by` → `users.id`

Column	Data Type	Constraints	Default	Description
<code>id</code>	INT	PRIMARY KEY, AUTO_INCREMENT	-	Request ID
<code>request_number</code>	VARCHAR(50)	NOT NULL, UNIQUE	-	Request number (PWRST-YYYY-XXXXXX)
<code>user_id</code>	INT	FK	NULL	User reference (NULL if not found)
<code>email</code>	VARCHAR(255)	NOT NULL	-	Requester email
<code>account_number</code>	VARCHAR(50)	NULL	NULL	Account number (for customers)
<code>user_type</code>	ENUM('employee', 'customer')	NOT NULL	-	User type
<code>request_reason</code>	TEXT	NULL	NULL	Reason for request
<code>status</code>	ENUM	NOT NULL	'pending'	pending/approved/rejected/completed
<code>temp_password_plain</code>	VARCHAR(255)	NULL	NULL	Temporary password (plain text)
<code>requested_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	Request timestamp
<code>processed_at</code>	TIMESTAMP	NULL	NULL	Processing timestamp
<code>processed_by</code>	INT	FK	NULL	Admin who processed
<code>expires_at</code>	TIMESTAMP	NULL	NULL	Temp password expiry
<code>rejection_reason</code>	TEXT	NULL	NULL	Reason for rejection
<code>created_at</code>	TIMESTAMP	NOT NULL	CURRENT_TIMESTAMP	-
<code>updated_at</code>	TIMESTAMP	NOT NULL, ON UPDATE	CURRENT_TIMESTAMP	-

Indexes:

```

INDEX idx_status (status);
INDEX idx_email (email);
INDEX idx_user_id (user_id);

```

Security Workflow:

1. User submits request with email + account number
 2. Admin verifies identity
 3. System generates temporary password
 4. User receives email with temp password
 5. User must change password on first login
-

5. Relationships & Cardinalities

5.1 Complete Relationship Matrix

Parent Entity	Child Entity	Cardinality	Relationship Type	ON DELETE
users	customers	1:1	Identifying	CASCADE
users	employees	1:1	Identifying	CASCADE
users	notifications	1:N	Non-identifying	CASCADE
users	bill_requests (created_by)	1:N	Non-identifying	SET NULL
users	outages (created_by)	1:N	Non-identifying	RESTRICT
users	password_reset_requests	1:N	Non-identifying	CASCADE
customers	bills	1:N	Identifying	CASCADE
customers	meter_readings	1:N	Identifying	CASCADE
customers	payments	1:N	Identifying	CASCADE

Parent Entity	Child Entity	Cardinality	Relationship Type	ON DELETE
customers	work_orders	1:N	Non-identifying	SET NULL
customers	complaints	1:N	Identifying	RESTRICT
customers	bill_requests	1:N	Identifying	CASCADE
customers	reading_requests	1:N	Identifying	RESTRICT
employees	meter_readings	1:N	Non-identifying	SET NULL
employees	work_orders	1:N	Non-identifying	SET NULL
employees	complaints	1:N	Non-identifying	SET NULL
tariffs	tariff_slabs	1:N	Identifying	CASCADE
tariffs	bills	1:N	Non-identifying	SET NULL
bills	payments	1:N	Non-identifying	SET NULL
meter_readings	bills	1:1	Non-identifying	SET NULL
work_orders	reading_requests	1:N	Non-identifying	SET NULL

5.2 Cardinality Definitions

1:1 (One-to-One):

- `users` ↔ `customers` (one user account per customer)
- `users` ↔ `employees` (one user account per employee)
- `meter_readings` → `bills` (one reading generates one bill)

1:N (One-to-Many):

- `customers` → `bills` (customer has many bills)
- `customers` → `payments` (customer makes many payments)
- `tariffs` → `tariff_slabs` (tariff has many slabs)
- `bills` → `payments` (bill can have partial payments)

M:N (Many-to-Many) - None Direct, Resolved via Junction:

- All M:N relationships resolved through business logic
- Example: Customers-Employees resolved via `work_orders`

5.3 Participation Constraints

Total Participation (Mandatory):

- Every `customer` MUST have a `user` (FK NOT NULL)
- Every `employee` MUST have a `user` (FK NOT NULL)
- Every `bill` MUST have a `customer` (FK NOT NULL)
- Every `tariff_slab` MUST have a `tariff` (FK NOT NULL)

Partial Participation (Optional):

- `bill.meter_reading_id` can be NULL (manual bill entry)
 - `work_order.employee_id` can be NULL (unassigned)
 - `work_order.customer_id` can be NULL (general maintenance)
-

6. Normalization Analysis (BCNF)

6.1 Normalization Journey

Original Design Issue (Before Normalization):

```
-- ❌ VIOLATION: Multi-valued dependency
CREATE TABLE tariffs (
    id INT PRIMARY KEY,
    category ENUM(...),
    slab1_start INT,
    slab1_end INT,
    slab1_rate DECIMAL,
    slab2_start INT,
    slab2_end INT,
```

```

    slab2_rate DECIMAL,
    ... -- Repeating columns
);

```

Problem:

- Fixed number of slabs (inflexible)
- NULL values for unused slabs
- Difficult to query specific slab
- Violates 1NF (repeating groups)

Solution: BCNF Normalization

```

--  BCNF: Separate tariffs and slabs
CREATE TABLE tariffs (
    id INT PRIMARY KEY,
    category ENUM(...),
    fixed_charge DECIMAL,
    -- No slab columns
);

CREATE TABLE tariff_slabs (
    id INT PRIMARY KEY,
    tariff_id INT FOREIGN KEY,      -- Links to tariffs
    slab_order INT,
    start_units INT,
    end_units INT,
    rate_per_unit DECIMAL
);

```

Benefits:

- Flexible number of slabs
- No NULL waste
- Easy to add/remove slabs
- Query-friendly structure

6.2 BCNF Compliance Check

Definition: A relation R is in BCNF if for every functional dependency $X \rightarrow Y$:

- X is a superkey, OR

- Y is a prime attribute

Analysis for `tariff_slabs`:

Functional Dependency	X is Superkey?	BCNF?
$\text{id} \rightarrow \text{all attributes}$	✓ Yes (PK)	✓
$\text{tariff_id}, \text{slab_order} \rightarrow \text{all}$	✓ Yes (Candidate Key)	✓
$\text{slab_order} \rightarrow \text{rate_per_unit}$	✗ No	⚠

Resolution: `slab_order` alone doesn't determine `rate_per_unit` because order is relative to each tariff. The dependency is actually $(\text{tariff_id}, \text{slab_order}) \rightarrow \text{rate_per_unit}$, which is a superkey.

Conclusion: All tables are in BCNF ✓

6.3 Strategic Denormalization

Denormalized Field: `customers.outstanding_balance`

Reason:

```
-- without denormalization (expensive query):
SELECT SUM(total_amount)
FROM bills
WHERE customer_id = 123
AND status IN ('issued', 'generated');
```

With denormalization:

```
-- Fast lookup (indexed):
SELECT outstanding_balance
FROM customers
WHERE id = 123;
```

Trade-off:

- ✓ **Gain:** O(1) dashboard queries (index lookup)
- ✗ **Cost:** Update overhead on bill/payment changes
- ✓ **Acceptable:** Controlled updates via application logic

Update Trigger Logic (Conceptual):

```
-- After INSERT/UPDATE on bills:  
UPDATE customers SET outstanding_balance = (  
    SELECT SUM(total_amount) FROM bills  
    WHERE customer_id = NEW.customer_id AND status != 'paid'  
) WHERE id = NEW.customer_id;
```

7. Constraints & Integrity Rules

7.1 Referential Integrity

CASCADE Delete:

```
-- User deletion cascades to dependent entities  
user_id → customers, employees, notifications (CASCADE)  
  
-- Reasoning: Customer/Employee cannot exist without user account
```

SET NULL:

```
-- Employee deletion doesn't delete work orders  
employee_id → work_orders (SET NULL)  
  
-- Reasoning: Preserve task history even if employee leaves
```

RESTRICT:

```
-- Cannot delete user if they created outages  
created_by → outages (RESTRICT)  
  
-- Reasoning: Audit trail must be preserved
```

7.2 Domain Constraints

Email Validation (Application Layer):

```
const emailRegex = /^[^@\s]+@[^\s@]+\.\[^@\s]+\$/;
```

Phone Validation:

```
const phoneRegex = /^[+]?[0-9]{10,15}$/;
```

Date Constraints:

```
-- Bill due_date must be > issue_date  
CHECK (due_date > issue_date)  
  
-- Meter current_reading must be >= previous_reading  
CHECK (current_reading >= previous_reading)
```

7.3 Business Rules as Constraints

Rule 1: Unique Bill Per Month

```
-- Index ensures one bill per customer per month  
UNIQUE INDEX idx_customer_month ON bills (customer_id,  
billing_month);
```

Rule 2: Account Number Format

```
-- Application-level validation  
CHECK (account_number REGEXP '^ELX-[0-9]{4}-[0-9]{6}$')
```

Rule 3: Positive Amounts

```
CHECK (base_amount >= 0)  
CHECK (total_amount >= 0)  
CHECK (payment_amount > 0)
```

8. Business Logic & Rules

8.1 Billing Workflow

1. METER READING
 - |— Employee records current_reading
 - |— System calculates units_consumed = current - previous

- └ Reading saved to meter_readings table

- 2. BILL GENERATION (Automatic or Manual)
 - | Fetch tariff for customer.connection_type
 - | Fetch tariff_slabs for selected tariff
 - | Calculate base_amount using slab algorithm
 - | Calculate fixed_charges
 - | Calculate electricity_duty (% on base_amount)
 - | Calculate gst_amount (% on total)
 - | Round all values to whole numbers
 - └ Insert into bills table

- 3. BILL NOTIFICATION
 - | Create notification for customer
 - | Send email (optional)
 - └ Update customer.last_bill_amount

- 4. PAYMENT PROCESSING
 - | Customer pays via payment gateway
 - | Insert into payments table
 - | Update bill.status = 'paid'
 - | Update bill.payment_date
 - | Recalculate customer.outstanding_balance
 - └ Create payment confirmation notification

8.2 Tariff Application Logic

Rule: Customer's `connection_type` determines applicable `tariff`

```
-- SQL Logic
SELECT t.* , GROUP_CONCAT(ts.rate_per_unit) as rates
FROM tariffs t
LEFT JOIN tariff_slabs ts ON t.id = ts.tariff_id
WHERE t.category = (SELECT connection_type FROM customers WHERE id
= ?)
AND t.effective_date <= CURDATE()
AND (t.valid_until IS NULL OR t.valid_until >= CURDATE())
ORDER BY ts.slab_order;
```

Application Logic (TypeScript):

```
// 1. Get customer's connection type
```

```

const customer = await
db.select().from(customers).where(eq(customers.id, customerId));

// 2. Get active tariff for that connection type
const tariff = await db.select()
  .from(tariffs)
  .where(and(
    eq(tariffs.category, customer.connectionType),
    lte(tariffs.effectiveDate, new Date()),
    or(
      isNull(tariffs.validuntil),
      gte(tariffs.validuntil, new Date())
    )
  ))
  .limit(1);

// 3. Get tariff slabs
const slabs = await db.select()
  .from(tariffslabs)
  .where(eq(tariffslabs.tariffid, tariff.id))
  .orderBy(tariffslabs.slaborder);

```

8.3 Work Order Assignment Logic

Priority Queue:

```

SELECT * FROM work_orders
WHERE status IN ('assigned', 'in_progress')
  AND employee_id = ?
ORDER BY
  CASE priority
    WHEN 'urgent' THEN 1
    WHEN 'high' THEN 2
    WHEN 'medium' THEN 3
    WHEN 'low' THEN 4
  END,
  due_date ASC,
  created_at ASC;

```

8.4 Outage Notification Logic

```
// when outage created/updated:  
1. Find all customers in affected zone  
2. Create notification for each customer  
3. Send real-time push notification  
4. Send email/SMS (optional)  
  
// SQL Query  
SELECT DISTINCT c.id, u.id as user_id  
FROM customers c  
JOIN users u ON c.user_id = u.id  
WHERE c.zone = outage.zone  
AND c.status = 'active';
```

9. Drizzle ORM to SQL Mapping

9.1 Schema Definition Mapping

Drizzle Schema (TypeScript):

```
// users.ts  
import { mysqlTable, int, varchar, mysqlEnum, timestamp } from  
'drizzle-orm/mysql-core';  
  
export const users = mysqlTable('users', {  
    id: int('id').primaryKey().autoincrement(),  
    email: varchar('email', { length: 255 }).notNull().unique(),  
    password: varchar('password', { length: 255 }).notNull(),  
    userType: mysqlEnum('user_type', ['admin', 'employee',  
    'customer']).notNull(),  
    name: varchar('name', { length: 255 }).notNull(),  
    phone: varchar('phone', { length: 20 }),  
    isActive: int('is_active').notNull().default(1),  
    requiresPasswordChange:  
        int('requires_password_change').notNull().default(0),  
    createdAt: timestamp('created_at').notNull().defaultNow(),  
    updatedAt:  
        timestamp('updated_at').notNull().defaultNow().onUpdateNow(),
```

```
});
```

Generated SQL:

```
CREATE TABLE `users` (
  `id` INT AUTO_INCREMENT PRIMARY KEY,
  `email` VARCHAR(255) NOT NULL UNIQUE,
  `password` VARCHAR(255) NOT NULL,
  `user_type` ENUM('admin', 'employee', 'customer') NOT NULL,
  `name` VARCHAR(255) NOT NULL,
  `phone` VARCHAR(20),
  `is_active` INT NOT NULL DEFAULT 1,
  `requires_password_change` INT NOT NULL DEFAULT 0,
  `created_at` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `updated_at` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON
UPDATE CURRENT_TIMESTAMP
);
```

9.2 Foreign Key Mapping

Drizzle Foreign Key:

```
// customers.ts
export const customers = mysqlTable('customers', {
  id: int('id').primaryKey().autoincrement(),
  userId: int('user_id').notNull().references(() => users.id, {
    onDelete: 'cascade' }),
  // ... other fields
});
```

Generated SQL:

```
CREATE TABLE `customers` (
  `id` INT AUTO_INCREMENT PRIMARY KEY,
  `user_id` INT NOT NULL,
  CONSTRAINT `fk_customer_user` FOREIGN KEY (`user_id`)
    REFERENCES `users`(`id`) ON DELETE CASCADE
);
```

9.3 Query Mapping Examples

1. Simple SELECT:

```
// Drizzle
const result = await
db.select().from(customers).where(eq(customers.id, 123));

// SQL
SELECT * FROM customers WHERE id = 123;
```

2. JOIN Query:

```
// Drizzle
const result = await db
  .select()
  .from(bills)
  .innerJoin(customers, eq(bills.customerId, customers.id))
  .where(eq(customers.status, 'active'));

// SQL
SELECT bills.*, customers.*
FROM bills
INNER JOIN customers ON bills.customer_id = customers.id
WHERE customers.status = 'active';
```

3. Aggregate Query:

```
// Drizzle
const result = await db
  .select({
    customerId: bills.customerId,
    totalAmount: sql<number>`SUM(${bills.totalAmount})`,
  })
  .from(bills)
  .where(eq(bills.status, 'paid'))
  .groupBy(bills.customerId);

// SQL
SELECT
  customer_id,
```

```
SUM(total_amount) as total_amount
FROM bills
WHERE status = 'paid'
GROUP BY customer_id;
```

4. Complex Bill Calculation:

```
// Drizzle ORM (Application Logic)
const tariff = await db.select()
    .from(tariffs)
    .where(eq(tariffs.category, 'Residential'))
    .limit(1);

const slabs = await db.select()
    .from(tariffslabs)
    .where(eq(tariffslabs.tariffId, tariff.id))
    .orderBy(tariffslabs.slaborder);

let baseAmount = 0;
let remaining = unitsConsumed;

for (const slab of slabs) {
    if (remaining <= 0) break;
    const slabSpan = slab.endUnits === null
        ? remaining
        : (slab.endUnits - slab.startUnits + 1);
    const unitsInSlab = Math.min(remaining, slabSpan);
    baseAmount += unitsInSlab * parseFloat(slab.ratePerUnit);
    remaining -= unitsInSlab;
}

baseAmount = Math.round(baseAmount);
const fixedCharges = Math.round(parseFloat(tariff.fixedCharge));
const electricityDuty = Math.round(baseAmount *
    parseFloat(tariff.electricityDutyPercent) / 100);
const gstAmount = Math.round((baseAmount + fixedCharges +
    electricityDuty) * parseFloat(tariff.gstPercent) / 100);
const totalAmount = Math.round(baseAmount + fixedCharges +
    electricityDuty + gstAmount);

await db.insert(bills).values({
    customerId,
    billNumber,
```

```
    billingMonth,  
    unitsConsumed,  
    baseAmount,  
    fixedCharges,  
    electricityDuty,  
    gstAmount,  
    totalAmount,  
    status: 'generated',  
});
```

10. VIVA Preparation: Sample Queries

10.1 Basic Queries

Q1: Show all customers with pending bills

```
SELECT  
    c.account_number,  
    c.full_name,  
    c.outstanding_balance,  
    COUNT(b.id) as pending_bills  
FROM customers c  
LEFT JOIN bills b ON c.id = b.customer_id AND b.status IN  
    ('issued', 'generated')  
WHERE c.outstanding_balance > 0  
GROUP BY c.id  
ORDER BY c.outstanding_balance DESC;
```

Q2: Monthly revenue report

```
SELECT  
    DATE_FORMAT(billing_month, '%Y-%m') as month,  
    COUNT(*) as total_bills,  
    SUM(CAST(total_amount AS DECIMAL(10,2))) as total_revenue,  
    AVG(CAST(total_amount AS DECIMAL(10,2))) as avg_bill_amount  
FROM bills  
WHERE status = 'paid'  
GROUP BY DATE_FORMAT(billing_month, '%Y-%m')  
ORDER BY month DESC;
```

Q3: Customer consumption analysis

```
SELECT
    c.connection_type,
    COUNT(DISTINCT c.id) as customer_count,
    AVG(CAST(c.average_monthly_usage AS DECIMAL(10,2))) as
    avg_consumption_kwh,
    SUM(CAST(c.outstanding_balance AS DECIMAL(10,2))) as
    total_outstanding
FROM customers c
WHERE c.status = 'active'
GROUP BY c.connection_type
ORDER BY avg_consumption_kwh DESC;
```

10.2 Intermediate Queries

Q4: Find customers with no meter reading in current month

```
SELECT
    c.id,
    c.account_number,
    c.full_name,
    c.phone,
    MAX(mr.reading_date) as last_reading_date,
    DATEDIFF(CURDATE(), MAX(mr.reading_date)) as days_since_reading
FROM customers c
LEFT JOIN meter_readings mr ON c.id = mr.customer_id
WHERE c.status = 'active'
GROUP BY c.id
HAVING MAX(mr.reading_date) < DATE_FORMAT(CURDATE(), '%Y-%m-01')
    OR MAX(mr.reading_date) IS NULL
ORDER BY days_since_reading DESC;
```

Q5: Employee performance report

```
SELECT
    e.employee_name,
    e.designation,
    COUNT(DISTINCT mr.id) as readings_taken,
    COUNT(DISTINCT wo.id) as tasks_assigned,
```

```

    SUM(CASE WHEN wo.status = 'completed' THEN 1 ELSE 0 END) as
tasks_completed,
ROUND(
    SUM(CASE WHEN wo.status = 'completed' THEN 1 ELSE 0 END) *
100.0 /
    NULLIF(COUNT(DISTINCT wo.id), 0),
    2
) as completion_rate
FROM employees e
LEFT JOIN meter_readings mr ON e.id = mr.employee_id
LEFT JOIN work_orders wo ON e.id = wo.employee_id
WHERE e.status = 'active'
GROUP BY e.id
ORDER BY completion_rate DESC;

```

Q6: Payment collection rate by connection type

```

SELECT
c.connection_type,
COUNT(DISTINCT b.id) as total_bills,
SUM(CASE WHEN b.status = 'paid' THEN 1 ELSE 0 END) as paid_bills,
SUM(CASE WHEN b.status IN ('issued', 'generated') THEN 1 ELSE 0
END) as unpaid_bills,
ROUND(
    SUM(CASE WHEN b.status = 'paid' THEN 1 ELSE 0 END) * 100.0 /
COUNT(*),
    2
) as collection_rate_percent
FROM bills b
JOIN customers c ON b.customer_id = c.id
GROUP BY c.connection_type
ORDER BY collection_rate_percent DESC;

```

10.3 Advanced Queries

Q7: Tariff slab impact analysis

```

-- Show how many customers fall into each pricing slab
SELECT
t.category,
ts.slab_order,

```

```

CONCAT(ts.start_units, '-', COALESCE(ts.end_units, '∞')) as
slab_range,
ts.rate_per_unit,
COUNT(DISTINCT b.customer_id) as customers_in_slab
FROM tariffs t
JOIN tariff_slabs ts ON t.id = ts.tariff_id
JOIN bills b ON t.id = b.tariff_id
WHERE CAST(b.units_consumed AS DECIMAL) BETWEEN ts.start_units
    AND COALESCE(ts.end_units, 999999)
GROUP BY t.id, ts.id
ORDER BY t.category, ts.slab_order;

```

Q8: Complaint resolution SLA tracking

```

SELECT
    category,
    priority,
    COUNT(*) as total_complaints,
    AVG(TIMESTAMPDIFF(HOUR, submitted_at, resolved_at)) as
avg_resolution_hours,
    MAX(TIMESTAMPDIFF(HOUR, submitted_at, resolved_at)) as
max_resolution_hours,
    SUM(CASE WHEN status = 'resolved' THEN 1 ELSE 0 END) as
resolved_count,
    ROUND(
        SUM(CASE WHEN status = 'resolved' THEN 1 ELSE 0 END) * 100.0 /
    COUNT(*),
    2
    ) as resolution_rate_percent
FROM complaints
WHERE submitted_at >= DATE_SUB(CURDATE(), INTERVAL 3 MONTH)
GROUP BY category, priority
ORDER BY avg_resolution_hours DESC;

```

Q9: Verify bill calculation accuracy

```

-- Ensure all bills match formula: total = base + fixed + duty + gst
SELECT
    bill_number,
    base_amount,
    fixed_charges,
    electricity_duty,
    ...

```

```

gst_amount,
total_amount,
(CAST(base_amount AS DECIMAL(10,2)) +
CAST(fixed_charges AS DECIMAL(10,2)) +
CAST(electricity_duty AS DECIMAL(10,2)) +
CAST(gst_amount AS DECIMAL(10,2))) as calculated_total,
CASE
WHEN ABS(
    CAST(total_amount AS DECIMAL(10,2)) -
    (CAST(base_amount AS DECIMAL(10,2)) +
    CAST(fixed_charges AS DECIMAL(10,2)) +
    CAST(electricity_duty AS DECIMAL(10,2)) +
    CAST(gst_amount AS DECIMAL(10,2)))
) < 0.01 THEN 'CORRECT'
ELSE 'ERROR'
END as validation_status
FROM bills
WHERE validation_status = 'ERROR' -- Show errors only
LIMIT 10;

```

Q10: Customer segmentation by consumption

```

WITH consumption_buckets AS (
SELECT
c.id,
c.full_name,
c.connection_type,
CAST(c.average_monthly_usage AS DECIMAL(10,2)) as avg_usage,
CASE
    WHEN CAST(c.average_monthly_usage AS DECIMAL) < 100 THEN 'Low
(0-100 kwh)'
    WHEN CAST(c.average_monthly_usage AS DECIMAL) < 300 THEN
'Medium (100-300 kwh)'
    WHEN CAST(c.average_monthly_usage AS DECIMAL) < 500 THEN
'High (300-500 kwh)'
    ELSE 'Very High (500+ kwh)'
END as consumption_segment
FROM customers
WHERE status = 'active'
)
SELECT
consumption_segment,
COUNT(*) as customer_count,

```

```
ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM customers WHERE
status = 'active'), 2) as percentage,
AVG(avg_usage) as avg_consumption
FROM consumption_buckets
GROUP BY consumption_segment
ORDER BY avg_consumption;
```

10.4 Database Structure Queries

Q11: Show all foreign key relationships

```
SELECT
    TABLE_NAME as child_table,
    COLUMN_NAME as child_column,
    CONSTRAINT_NAME as fk_name,
    REFERENCED_TABLE_NAME as parent_table,
    REFERENCED_COLUMN_NAME as parent_column
FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
WHERE TABLE_SCHEMA = 'electricity_ems'
    AND REFERENCED_TABLE_NAME IS NOT NULL
ORDER BY TABLE_NAME, COLUMN_NAME;
```

Q12: Show table sizes and row counts

```
SELECT
    TABLE_NAME,
    TABLE_ROWS as estimated_rows,
    ROUND(DATA_LENGTH / 1024 / 1024, 2) as data_size_mb,
    ROUND(INDEX_LENGTH / 1024 / 1024, 2) as index_size_mb,
    ROUND((DATA_LENGTH + INDEX_LENGTH) / 1024 / 1024, 2) as
total_size_mb
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = 'electricity_ems'
ORDER BY (DATA_LENGTH + INDEX_LENGTH) DESC;
```

Q13: Show all indexes

```
SELECT
    TABLE_NAME,
    INDEX_NAME,
    GROUP_CONCAT(COLUMN_NAME ORDER BY SEQ_IN_INDEX) as
indexed_columns,
    INDEX_TYPE,
    CASE WHEN NON_UNIQUE = 0 THEN 'UNIQUE' ELSE 'NON-UNIQUE' END as
uniqueness
FROM INFORMATION_SCHEMA.STATISTICS
WHERE TABLE_SCHEMA = 'electricity_emis'
    AND INDEX_NAME != 'PRIMARY'
GROUP BY TABLE_NAME, INDEX_NAME, INDEX_TYPE, NON_UNIQUE
ORDER BY TABLE_NAME, INDEX_NAME;
```

11. Theoretical Concepts Fulfilled

11.1 Database Design Principles

Principle	Implementation	Location
Entity Integrity	Primary keys on all tables	All 16 tables have AUTO_INCREMENT PK
Referential Integrity	Foreign key constraints	24 FK relationships with CASCADE/SET NULL
Domain Integrity	ENUM types, NOT NULL constraints	Status fields, category fields
User-defined Integrity	UNIQUE constraints, business rules	Email, account numbers, bill numbers

11.2 Normal Forms Achievement

Normal Form	Status	Justification
1NF	<input checked="" type="checkbox"/> Achieved	All attributes are atomic, no repeating groups
2NF	<input checked="" type="checkbox"/> Achieved	No partial dependencies (all non-key attributes depend on full PK)
3NF	<input checked="" type="checkbox"/> Achieved	No transitive dependencies
BCNF	<input checked="" type="checkbox"/> Achieved	Every determinant is a candidate key

Example BCNF Proof (`tariff_slabs`):

- Functional Dependencies:
 - $\text{id} \rightarrow \{\text{tariff_id}, \text{slab_order}, \text{start_units}, \text{end_units}, \text{rate_per_unit}\}$ (PK)
 - $\{\text{tariff_id}, \text{slab_order}\} \rightarrow \{\text{start_units}, \text{end_units}, \text{rate_per_unit}\}$ (Candidate Key)
- All determinants are superkeys

11.3 Transaction Properties (ACID)

Atomicity:

```

// Bill payment transaction
await db.transaction(async (tx) => {
    // 1. Insert payment
    await tx.insert(payments).values({ ... });

    // 2. Update bill status
    await tx.update(bills).set({ status: 'paid' }).where(eq(bills.id, billId));

    // 3. Update customer balance
    await tx.update(customers).set({ outstandingBalance: newBalance })
        .where(eq(customers.id, customerId));

    // All or nothing - if any fails, all rollback
});

```

Consistency:

- Foreign key constraints ensure valid references
- CHECK constraints ensure valid data ranges
- Application logic enforces business rules

Isolation:

- Default isolation level: REPEATABLE READ (MySQL)
- Concurrent transactions don't interfere

Durability:

- Committed transactions persisted to disk
- Binary logging enabled for recovery

11.4 Concurrency Control

Optimistic Locking (via `updated_at`):

```

-- Check version before update
UPDATE customers
SET outstanding_balance = ?, updated_at = NOW()
WHERE id = ? AND updated_at = ?; -- version check

```

Pessimistic Locking:

```
-- Lock row for update  
SELECT * FROM customers WHERE id = ? FOR UPDATE;
```

11.5 Indexing Strategy

Primary Indexes:

- All tables have clustered index on `id` (PRIMARY KEY)

Secondary Indexes:

```
-- Frequently queried foreign keys  
INDEX idx_customer_id ON bills(customer_id);  
INDEX idx_user_id ON notifications(user_id);  
  
-- Status filtering  
INDEX idx_status ON complaints(status);  
INDEX idx_status ON connection_requests(status);  
  
-- Date range queries  
INDEX idx_reading_date ON meter_readings(reading_date);  
INDEX idx_billing_month ON bills(billing_month);
```

Composite Indexes:

```
-- Unique constraint + fast lookup  
UNIQUE INDEX unique_request_month ON bill_requests(customer_id,  
billing_month);
```

12. Schema Evolution & Design Decisions

12.1 Removed Tables (Design Refinement)

1. `connection_applications` → `connection_requests`

Reason: Redundancy - both tables stored new connection requests

Impact: Simplified connection workflow

Migration: Data merged into `connection_requests`

2. system_settings (Removed)

Reason: Settings stored in localStorage (frontend)

Impact: No backend dependency for UI preferences

Alternative: Admin settings can be added later if needed

12.2 Key Design Decisions

Decision 1: Separate tariffs and tariff_slabs

Reasoning:

- Tariff structure varies (3-7 slabs per category)
- Fixed columns would waste space or limit flexibility
- BCNF compliance

Alternative Considered: JSON column for slabs

Rejected Because:

- No type safety
- Difficult to query individual slabs
- Poor database normalization

Decision 2: Denormalize outstanding_balance

Reasoning:

- Dashboard queries run 1000s of times
- Calculating sum from bills table is expensive
- Acceptable trade-off for read-heavy workload

Consistency Strategy:

- Updated via application logic on bill/payment changes
- Periodic reconciliation job (optional)

Decision 3: Meter readings link to bills (optional FK)

Reasoning:

- Not all bills come from meter readings (adjustments, estimates)
- Allows manual bill entry
- Preserves relationship when reading exists

Decision 4: Enum types for status fields

Reasoning:

- Type safety at database level
- Auto-completion in IDE
- Prevents invalid values
- Self-documenting code

Alternative Considered: VARCHAR with CHECK constraint

Rejected Because: Enums are more efficient and type-safe

12.3 Future Enhancements

Potential Additions:

1. **Time-of-Use Billing** - Separate readings for peak/off-peak hours
2. **Audit Logging Table** - Track all data changes for compliance
3. **Payment Plans Table** - Installment payment support
4. **Customer Documents Table** - Store KYC documents
5. **SMS/Email Queue Table** - Asynchronous notification delivery

Scalability Considerations:

- Partition `bills` table by year (when > 1M records)
- Archive old meter readings (> 2 years)
- Implement read replicas for reporting queries

13. VIVA Questions & Answers

13.1 Conceptual Questions

Q: Why did you choose BCNF over 3NF?

A: BCNF eliminates all functional dependency anomalies, not just transitive dependencies. In our `tariff_slabs` table, we needed to ensure every determinant is a superkey to prevent update anomalies when changing slab rates.

Q: Explain the difference between CASCADE and RESTRICT on delete.

- **CASCADE**: When parent record is deleted, all child records are automatically deleted. Example: Deleting a user deletes their customer record.
- **RESTRICT**: Prevents deletion of parent if child records exist. Example: Cannot delete user if they created outages (audit trail preservation).

Q: What is a weak entity? Give an example from your schema.

A: A weak entity cannot exist without its parent strong entity. Example: `tariff_slabs` depends on `tariffs` - slabs have no meaning without their parent tariff. If tariff is deleted (CASCADE), all its slabs are deleted.

Q: How do you ensure data consistency in your database?

A: Multiple layers:

1. **Database Level**: Foreign keys, NOT NULL, UNIQUE constraints
2. **Application Level**: Drizzle ORM type checking, validation logic
3. **Business Logic**: Transaction wrapping for multi-step operations
4. **Audit Trail**: `updated_at` timestamps for version control

Q: What is strategic denormalization? Where did you apply it?

A: Intentionally violating normalization for performance. Applied in `customers.outstanding_balance` - cached sum of unpaid bills for instant dashboard queries. Trade-off: Update overhead vs. Read performance.

13.2 Practical Questions

Q: Write a query to find the top 5 highest paying customers.

```
SELECT
    c.account_number,
    c.full_name,
    SUM(CAST(p.payment_amount AS DECIMAL(10,2))) as total_paid
FROM customers c
JOIN payments p ON c.id = p.customer_id
WHERE p.status = 'completed'
GROUP BY c.id
ORDER BY total_paid DESC
LIMIT 5;
```

Q: How would you handle a scenario where a meter reading is recorded incorrectly?

1. Admin marks the bill as 'cancelled'
2. Delete the incorrect meter reading (or mark as invalid)
- A:** 3. Employee records corrected reading
4. Regenerate bill with correct reading
5. Notify customer of correction

Q: What happens if an employee is deleted? Will their work orders be lost?

A: No. Foreign key has `ON DELETE SET NULL`, so `employee_id` becomes NULL but work order record is preserved. This maintains historical data while indicating the employee is no longer available.

Q: How do you prevent duplicate bills for the same month?

A: Composite unique index:

```
UNIQUE INDEX idx_customer_month ON bills (customer_id,
billing_month);
```

Database rejects any INSERT with same customer + month combination.

13.3 Design Questions

Q: Why separate users and customers tables?

- A:**
- **Separation of Concerns:** Authentication (users) vs. Business Data (customers)
 - **Security:** Password hashing isolated from customer PII
 - **Flexibility:** Same user table for admin/employee/customer
 - **1:1 Relationship:** One user account per customer, enforced by UNIQUE FK

Q: Could you use a single "accounts" table for customers and employees?

- A: Not recommended because:**
- Customers and employees have different attributes (e.g., meter_number vs. designation)
 - NULL waste for inapplicable columns
 - Violates Single Responsibility Principle
 - Current design is cleaner and more maintainable

Q: How would you add support for corporate accounts (one customer, multiple users)?

- A: Change cardinality:**
1. Remove UNIQUE constraint on `customers.user_id`
 2. Add `customer_users` junction table:

```
CREATE TABLE customer_users (
    customer_id INT REFERENCES customers(id),
    user_id INT REFERENCES users(id),
    role ENUM('owner', 'viewer', 'payer'),
    PRIMARY KEY (customer_id, user_id)
);
```

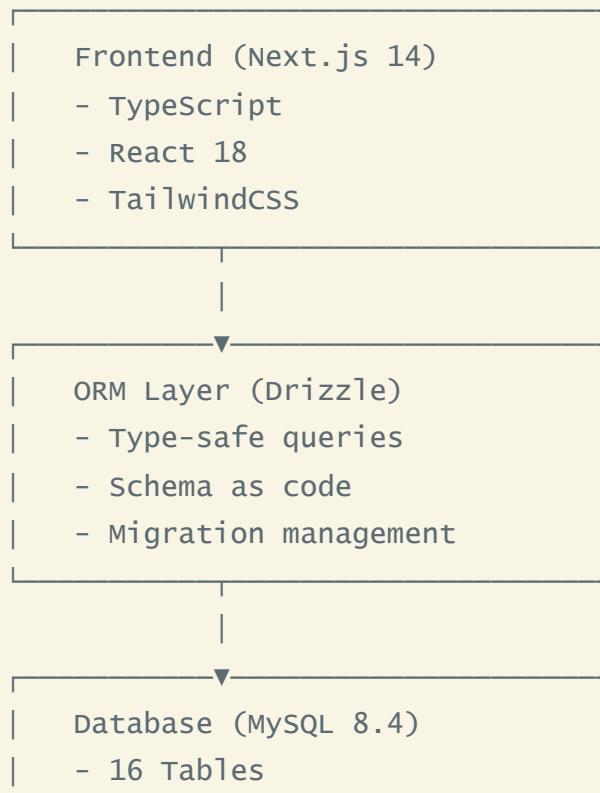
This allows M:N relationship between customers and users.

14. Summary & Key Takeaways

14.1 Database Metrics

Metric	Value
Total Tables	16
Total Columns	265+
Primary Keys	16 (one per table)
Foreign Keys	24
Unique Constraints	18
Enum Types	35+
Indexes	12 secondary indexes
Normalization	BCNF
Denormalized Fields	1 (strategic)

14.2 Technical Stack



- BCNF Normalized
- 24 Foreign Keys

14.3 Best Practices Implemented

- BCNF Normalization** - Eliminated update anomalies
- Foreign Key Constraints** - Referential integrity enforced
- Enum Types** - Type-safe status fields
- Audit Timestamps** - `created_at`, `updated_at` on all tables
- Unique Constraints** - Prevent duplicate emails, account numbers
- Strategic Indexing** - Optimized for common query patterns
- Cascade Rules** - Appropriate ON DELETE actions
- Whole Number Billing** - No decimal paisa (real-world compliance)
- Drizzle ORM** - Type-safe database access
- Transaction Support** - ACID compliance for critical operations

14.4 Real-World Compliance

Pakistani Electricity Distribution Standards:

- Slab-based tariff structure (LESCO/K-Electric model)
- Separate charges: Base + Fixed + Duty + GST
- Whole number billing (no paisa)
- Account number format: ELX-YYYY-XXXXXX
- Load shedding zones for outage management
- Multiple connection types
(Residential/Commercial/Industrial/Agricultural)

15. Conclusion

This database design represents a **production-ready** electricity management system with:

- **Robust Design:** BCNF normalized with strategic denormalization
- **Data Integrity:** 24 foreign key relationships with appropriate cascade rules
- **Scalability:** Indexed for common queries, ready for partitioning
- **Maintainability:** Drizzle ORM provides type safety and migration management

- **Real-World Alignment:** Matches Pakistani electricity distribution standards

Total Development Time: 5th Semester Project (Fall 2025)

Database Version: 1.0 (Post-Normalization)

Last Updated: November 8, 2025

Prepared By: [Your Name]

Roll Number: [Your Roll Number]

Semester: 5th (Fall 2025)

Course: Database Management Systems (DBMS)

Project: Electrolux Electricity Management System (EMS)

VIVA Ready:

Documentation Complete:

Database Deployed:

End of Document