

GenChargePhase2 - Database Design Specification

Database Design Specification (DDS)

GenCharge – Mobile Recharge System

Document Version: 2.0

Date: November 10, 2025

Prepared By: Saravanan B S

1. INTRODUCTION

1.1 Purpose

This Database Design Specification (DDS) document outlines the detailed structure and design considerations for GenCharge a mobile recharge system. The platform manages user accounts, transactions, plans, offers, notifications, and administrative functionalities to support prepaid and postpaid services, wallet management, referrals, and automated workflows. The system's primary datastore is PostgreSQL (~80% of data/workload) with MongoDB (~20%) used for audit logging, reporting, content management, and backup-related operations. This document serves as a comprehensive guide for developers, database administrators, and stakeholders to understand the database architecture, relationships, and constraints.

1.2 Scope

The telecommunications and user management application's database supports:

- Secure management of user accounts (prepaid and postpaid users, administrators)
- User authentication and session management
- Role-based access control for users and administrators
- Transaction processing for wallet top-ups, recharges, refunds, and referral rewards
- Management of plans and offers, including validity, criteria, and status

- Automated notifications and reminders for billing, plan subscriptions, and system updates
- Backup and restore operations with scheduling and logging
- Audit logging for tracking system actions (inserts, updates, deletes)
- Content management for FAQs, privacy policies, terms of service, and other static content
- Reporting and analytics for user activity, transactions, plans, offers, and referrals
- Metadata collections/tables for enumerations and lookup values (e.g., plan groups, offer types, notification types)

1.3 Definitions and Acronyms

- **DDS:** Database Design Specification
 - **PK:** Primary Key (PostgreSQL)
 - **FK:** Foreign Key (PostgreSQL)
 - **_id:** MongoDB ObjectID for primary keys in MongoDB collections
 - **TTL:** Time To Live (for session or temporary data expiration)
 - **ENUM:** Enumerated type used in PostgreSQL for predefined values (e.g., user_type, status)
 - **JSON:** JavaScript Object Notation used for flexible data storage in both PostgreSQL and MongoDB
 - **UUID:** Universally Unique Identifier used for session identification
-

2. DATABASE ARCHITECTURE

2.1 Database Management Systems Used

- **Architecture:** Polyglot Persistence
- **Primary DBMS:** PostgreSQL 15+ (for ≈80% of data)
- **Secondary DBMS:** MongoDB 3+ (for ≈20% of data)
- **Character Set (PostgreSQL):** UTF-8

- **Collation (PostgreSQL):** utf8_general_ci or locale-specific collation based on deployment requirements

2.2 Why Hybrid?

- **PostgreSQL** is ideal for relational data requiring transactional consistency, complex joins, and analytical queries, such as user management (`Users` , `Admins`), transactions (`Transactions`), plans (`Plans` , `CurrentActivePlans`), and (`BillReminders`).
- **MongoDB** is suited for flexible, schema-less data storage, such as audit logs (`AuditLog`), content management (`Content`), and backup/restore operations (`Notifications` , `Backup` , `BackupRestoreLogs` , `ReportsLogs`), where JSON documents and embedded structures enhance querying efficiency.
- A hybrid approach enables optimized performance by leveraging PostgreSQL's relational integrity for core operations and MongoDB's flexibility for logging, reporting, and unstructured data.

2.3 Naming Conventions

- **Table/Collection Names:** PascalCase with underscores (e.g., `Users` , `RolePermissions` , `BackupRestoreLogs`)
- **Field/Column Names:** lowercase with underscores (e.g., `user_id` , `refresh_token` , `backup_status`)
- **Primary Keys:**
 - PostgreSQL: `id` , `user_id` , `plan_id` , etc. (serial/integer or UUID for `Sessions.session_id`)
 - MongoDB: `audit_id` , `log_id` , `id` (ObjectID for collections like `AuditLog` , `ContactForm`)
- **Foreign Keys:** `referenced_table_id` or context-specific names (e.g., `user_id` , `role_id` , `plan_id`) to indicate relationships
- **ENUM Fields:** Descriptive names with ENUM constraints in PostgreSQL (e.g., `user_type` , `status` , `txn_type`)

2.4 Operational Considerations

- Use MongoDB TTL indexes for ephemeral data, such as session refresh tokens (`Sessions.refresh_token_expires_at`) and temporary backup logs (`BackupRestoreLogs`).
- Index frequently queried fields in PostgreSQL, such as `Users.user_id` , `Transactions.user_id` , `Plans.plan_id` , and date fields (`created_at` , `valid_to`) for performance optimization.

- Apply column-level encryption in PostgreSQL for sensitive fields (e.g., `Users.email`, `Users.phone_number`) and field-level encryption in MongoDB for sensitive JSON data (e.g., `Backup.details`, `AuditLog.details`).
- Ensure secure connections using TLS/SSL, implement VPC peering, enforce network policies, and apply least privilege access for all database interactions.

2.5 Additional Information

- Use MongoDB's capability for JSON documents in collections like `Content`, `Backup`, and `ReportsLogs` to store flexible metadata, criteria, and details (e.g., `Content.body`, `Backup.details`).
- Leverage PostgreSQL for enforcing strong data consistency in tables like `Users`, `Transactions`, and `Plans`, supporting complex joins and analytical queries for reporting (e.g., user transaction history, plan popularity).
- Maintain consistent referencing across DBMS boundaries by using clear field names (e.g., `user_id` in PostgreSQL aligns with `user_id` in MongoDB's `AuditLog`).
- Adopt appropriate indexes on foreign key columns in PostgreSQL (e.g., `Transactions.user_id`, `Plans.group_id`) to optimize query performance.

3. ENTITY-RELATIONSHIP

3.1 Entity Overview

PostgreSQL (80%) — Relational / Transaction-Critical Data

- **Users:** Stores user profiles, including name, email, phone number, referral details, user type (prepaid, postpaid), and wallet balance.
- **UsersArchive:** Archives user data with the same structure as `Users` for historical or deactivated accounts.
- **Sessions:** Manages active user sessions, including refresh tokens and session status.
- **Admins:** Stores admin profiles, including name, email, phone number, and status (active, blocked, deleted).
- **Roles:** Defines roles for users and admins (e.g., customer, admin, support).

- **Permissions:** Defines individual access rights (e.g., read, write, delete, edit) for resources.
- **RolePermissions:** Many-to-many join between **Roles** and **Permissions**.
- **Transactions:** Records financial transactions (e.g., wallet top-ups, recharges, refunds) with details like amount, payment method, and status.
- **Plans:** Stores plan details, including name, validity, type (prepaid, postpaid), and criteria.
- **Offers:** Manages promotional offers, including name, validity, type, and criteria.
- **CurrentActivePlans:** Tracks active plans assigned to users, including validity dates and status.
- **ReferralRewards:** Records rewards for referrals, including referrer, referred user, and reward amount.
- **AutoPay:** Manages automatic payment settings for users, including plan and due date.
- **BillReminders:** Tracks billing reminders for users, including due date and status.
- **PlanGroups:** Defines categories for plans (e.g., talktime, unlimited, roaming).
- **OfferTypes:** Defines types of offers (e.g., cashback, bonus_data, festive_special).
- **Backup:** Stores backup metadata, including snapshot name, storage URI, and status.
- **Schedule:** Manages backup schedules, including task name and next run time.

MongoDB (20%) — Flexible / Document-Oriented Data

- **AuditLog:** Logs system actions (insert, update, delete) with details stored in JSON.
- **ContactForm:** Stores user-submitted contact form data, including name, email, and description.
- **BackupRestoreLogs:** Logs backup and restore operations, including job type, status, and details.
- **Content:** Manages content like FAQs, privacy policies, and terms of service with JSON body.
- **Notifications:** Stores notification details, including sender, recipient, type, and status.

Utility Tables/Collections

- **PlanGroups** (PostgreSQL): Metadata for plan categories.

- **OfferTypes** (PostgreSQL): Metadata for offer types.

3.2 Key Relationships

Users

- One **Users** record can have many **Sessions** (1:M, via **user_id**).
- One **Users** record can have many **Transactions** (1:M, via **user_id**).
- One **Users** record can have many **CurrentActivePlans** (1:M, via **user_id**).
- One **Users** record can have many **Notifications** as recipient (1:M, via **recipient_id**).
- One **Users** record can have many **ReferralRewards** as referrer or referred (1:M, via **referrer_id** or **referred_id**).
- One **Users** record can have one **Roles** record (1:1, via **role_id**).
- One **Users** record can reference another **Users** record via **referred_by** to **referral_code** (self-referential, 1:1).
- One **Users** record can have many **AutoPay** settings (1:M, via **user_id**).
- One **Users** record can have many **BillReminders** (1:M, via **user_id**).
- One **Users** record can have many **AuditLog** entries in MongoDB (1:M, via **user_id**).

UsersArchive

- One **UsersArchive** record can reference one **Users.referral_code** via **referred_by** (1:1).
- One **UsersArchive** record can have one **Roles** record (1:1, via **role_id**).

Admins

- One **Admins** record can have one **Roles** record (1:1, via **role_id**).
- One **Admins** record can create many **Plans** (1:M, via **created_by**).
- One **Admins** record can create many **Offers** (1:M, via **created_by**).
- One **Admins** record can send many **Notifications** (1:M, via **sender_id**).
- One **Admins** record can create many **Content** records (1:M, via **created_by**).
- One **Admins** record can initiate many **Backup** records (1:M, via **created_by**).
- One **Admins** record can perform many **BackupRestoreLogs** actions (1:M, via **action_by**).
- One **Admins** record can generate many **ReportsLogs** (1:M, via **generated_by**).

Roles

- One `Roles` record can have many `Users` (1:M, via `role_id`).
- One `Roles` record can have many `Admins` (1:M, via `role_id`).
- One `Roles` record can have many `RolePermissions` entries (1:M, via `role_id`).

Permissions

- One `Permissions` record can have many `RolePermissions` entries (1:M, via `permission_id`).

RolePermissions

- One `RolePermissions` record links one `Roles` record to one `Permissions` record (M:N, via `role_id` and `permission_id`).

Transactions

- One `Transactions` record is associated with one `Users` record (1:1, via `user_id`).
- One `Transactions` record can reference one `Plans` record (1:1, via `plan_id`).
- One `Transactions` record can reference one `Offers` record (1:1, via `offer_id`).

Plans

- One `Plans` record can have many `CurrentActivePlans` (1:M, via `plan_id`).
- One `Plans` record can have many `Transactions` (1:M, via `plan_id`).
- One `Plans` record is associated with one `PlanGroups` record (1:1, via `group_id`).
- One `Plans` record is created by one `Admins` record (1:1, via `created_by`).
- One `Plans` record can have many `AutoPay` settings (1:M, via `plan_id`).
- One `Plans` record can have many `BillReminders` (1:M, via `plan_id`).

Offers

- One `Offers` record can have many `Transactions` (1:M, via `offer_id`).
- One `Offers` record is associated with one `OfferTypes` record (1:1, via `offer_type_id`).
- One `Offers` record is created by one `Admins` record (1:1, via `created_by`).

CurrentActivePlans

- One `CurrentActivePlans` record is associated with one `Users` record (1:1, via `user_id`).

- One `CurrentActivePlans` record is associated with one `Plans` record (1:1, via `plan_id`).

Notifications

- One `Notifications` record is sent by one `Admins` record or system (1:1, via `sender_id`).
- One `Notifications` record is associated with one `Users` record as recipient (1:1, via `recipient_id`).
- One `Notifications` record is associated with one `NotificationRecipientTypes` record (1:1, via `recipient_type_id`).
- One `Notifications` record is associated with one `NotificationTypes` record (1:1, via `type_id`).

Content

- One `Content` record is associated with one `ContentTypes` record (1:1, via `content_type_id`).
- One `Content` record is created by one `Admins` record (1:1, via `created_by`).

ReferralRewards

- One `ReferralRewards` record is associated with one `Users` record as referrer (1:1, via `referrer_id`).
- One `ReferralRewards` record is associated with one `Users` record as referred (1:1, via `referred_id`).

Backup

- One `Backup` record is created by one `Admins` record (1:1, via `created_by`).
- One `Backup` record can have many `Schedule` records (1:M, via `backup_id`).
- One `Backup` record can have many `BackupRestoreLogs` records (1:M, via `job_id`).

Schedule

- One `Schedule` record is associated with one `Backup` record (1:1, via `backup_id`).

BackupRestoreLogs

- One `BackupRestoreLogs` record is associated with one `Backup` record (1:1, via `job_id`).
- One `BackupRestoreLogs` record is performed by one `Admins` record (1:1, via `action_by`).

ReportsLogs

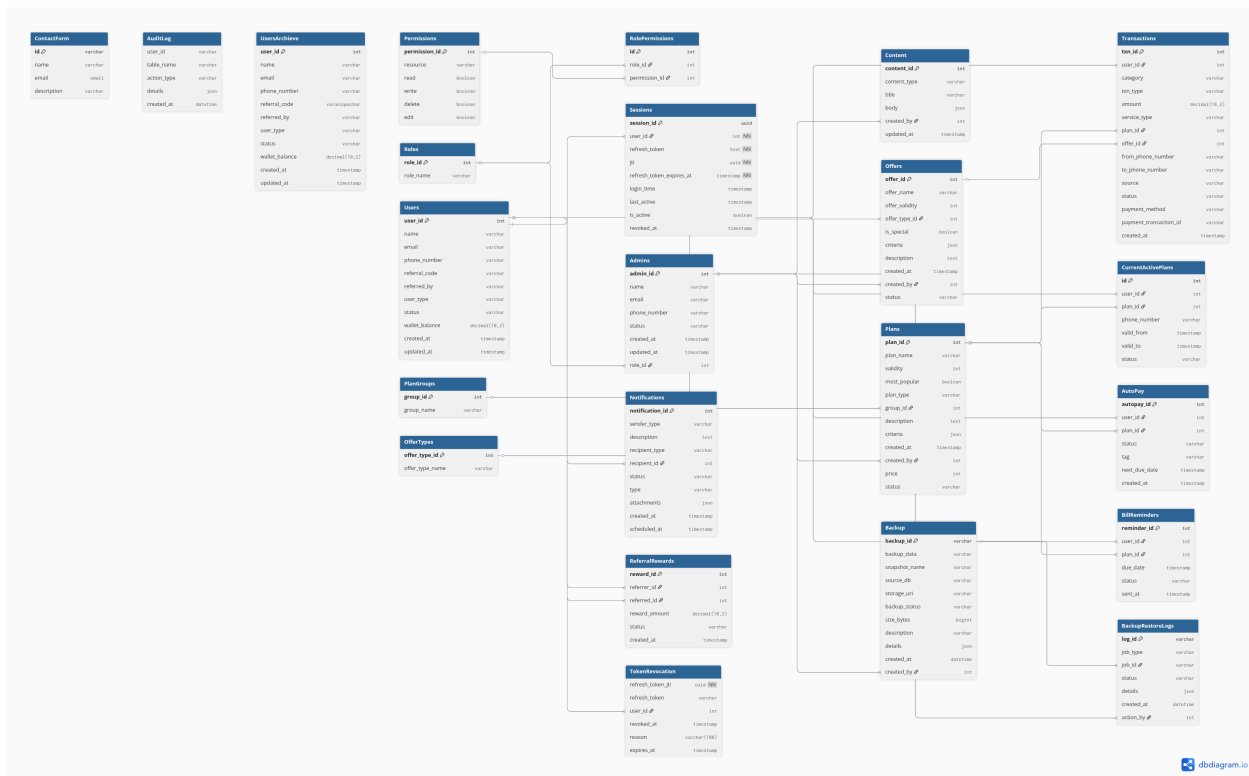
- One `ReportsLogs` record is associated with one `ReportTypes` record (1:1, via `report_type_id`).

- One **ReportsLogs** record is generated by one **Admins** record (1:1, via **generated_by**).

Utility Tables/Collections

- One **PlanGroups** record can have many **Plans** (1:M, via **group_id**).
- One **OfferTypes** record can have many **Offers** (1:M, via **offer_type_id**).
- One **NotificationRecipientTypes** record can have many **Notifications** (1:M, via **recipient_type_id**).
- One **NotificationTypes** record can have many **Notifications** (1:M, via **type_id**).
- One **ContentTypes** record can have many **Content** records (1:M, via **content_type_id**).
- One **ReportTypes** record can have many **ReportsLogs** (1:M, via **report_type_id**).

3.3 Entity – Relationship Diagram - view ER diagram



4. COLLECTION/TABLE SPECIFICATIONS

4.1 PostgreSQL Tables

Users Table

Purpose: Stores user profiles, including personal details, referral information, and wallet balance for prepaid and postpaid users.

Column Name	Data Type	Constraints	Description
user_id	INT	PK	Unique user identifier
name	VARCHAR		User's full name
email	VARCHAR	UNIQUE	User's email address
phone_number	VARCHAR	UNIQUE	User's phone number
referral_code	VARCHAR	UNIQUE	Unique referral code for the user
referred_by	VARCHAR	FK → Users.referral_code	Referral code of the referring user
user_type	VARCHAR	ENUM(prepaid, postpaid)	Type of user account
status	VARCHAR	ENUM(active, blocked)	Account status
wallet_balance	DECIMAL(10,2)	DEFAULT 0	User's wallet balance
created_at	TIMESTAMP		Record creation timestamp
updated_at	TIMESTAMP		Last update timestamp
role_id	INT	FK → Roles.role_id	Associated role identifier

UsersArchive Table

Purpose: Archives user data for historical or deactivated accounts.

Column Name	Data Type	Constraints	Description
user_id	INT	PK	Unique user identifier
name	VARCHAR		User's full name
email	VARCHAR	UNIQUE	User's email address
phone_number	VARCHAR	UNIQUE	User's phone number
referral_code	VARCHAR	UNIQUE	Unique referral code for the user
referred_by	VARCHAR	FK → Users.referral_code	Referral code of the referring user
user_type	VARCHAR	ENUM(prepaid, postpaid)	Type of user account
status	VARCHAR	ENUM(active, blocked)	Account status
wallet_balance	DECIMAL(10,2)	DEFAULT 0	User's wallet balance
created_at	TIMESTAMP		Record creation timestamp
updated_at	TIMESTAMP		Last update timestamp
role_id	INT	FK → Roles.role_id	Associated role identifier

Sessions Table

Purpose: Manages active user sessions and refresh tokens.

Column Name	Data Type	Constraints	Description
session_id	UUID	PK	Unique session identifier
user_id	INT	FK → Users.user_id, NOT NULL, CASCADE	Linked user identifier
refresh_token	TEXT	UNIQUE, NOT NULL	Session refresh token
refresh_token_expires_at	TIMESTAMP	NOT NULL	Token expiry timestamp
login_time	TIMESTAMP		Session login timestamp
last_active	TIMESTAMP		Last active timestamp
is_active	BOOLEAN	DEFAULT true	Session active status
revoked_at	TIMESTAMP		Session revocation timestamp

Admins Table

Purpose: Stores administrator account details.

Column Name	Data Type	Constraints	Description
admin_id	INT	PK	Unique admin identifier
name	VARCHAR		Admin's full name
email	VARCHAR	UNIQUE	Admin's email address
phone_number	VARCHAR	UNIQUE	Admin's phone number
status	VARCHAR	ENUM(active, blocked, deleted)	Admin account status
created_at	TIMESTAMP		Record creation timestamp
updated_at	TIMESTAMP		Last update timestamp
role_id	INT	FK → Roles.role_id	Associated role identifier

Roles Table

Purpose: Defines roles within the system.

Column Name	Data Type	Constraints	Description
role_id	INT	PK	Unique role identifier
role_name	VARCHAR		Role name (e.g., Customer, Admin)

Permissions Table

Purpose: Defines access rights or permissions for resources.

Column Name	Data Type	Constraints	Description
permission_id	INT	PK	Unique permission identifier
resource	VARCHAR		Permission name (e.g., view_plans)
read	BOOLEAN		Read permission flag
write	BOOLEAN		Write permission flag
delete	BOOLEAN		Delete permission flag
edit	BOOLEAN		Edit permission flag

RolePermissions Table

Purpose: Links roles to multiple permissions (many-to-many).

Column Name	Data Type	Constraints	Description
id	INT	PK	Unique record identifier
role_id	INT	FK → Roles.role_id	Associated role identifier
permission_id	INT	FK → Permissions.permission_id	Associated permission identifier

Transactions Table

Purpose: Records financial transactions, such as recharges and wallet top-ups.

Column Name	Data Type	Constraints	Description
txn_id	INT	PK	Unique transaction identifier
user_id	INT	FK → Users.user_id	Linked user identifier
category	VARCHAR	ENUM(wallet, service)	Transaction category
txn_type	VARCHAR	ENUM(credit, debit)	Transaction type
amount	DECIMAL(10,2)		Transaction amount

Column Name	Data Type	Constraints	Description
service_type	VARCHAR	ENUM(prepaid, postpaid)	Service type
plan_id	INT	FK → Plans.plan_id	Associated plan identifier
offer_id	INT	FK → Offers.offer_id	Associated offer identifier
from_phone_number	VARCHAR		Source phone number
to_phone_number	VARCHAR		Destination phone number
source	VARCHAR	ENUM(recharge, wallet_topup, refund, referral_reward, autopay)	Transaction source
status	VARCHAR	ENUM(success, failed, pending)	Transaction status
payment_method	VARCHAR	ENUM(UPI, Card, NetBanking, Wallet)	Payment method
payment_transaction_id	VARCHAR		External payment transaction ID
created_at	TIMESTAMP		Record creation timestamp

Plans Table

Purpose: Stores details of available plans for users.

Column Name	Data Type	Constraints	Description
plan_id	INT	PK	Unique plan identifier
plan_name	VARCHAR		Plan name
validity	INT		Plan validity period (days)
most_popular	BOOLEAN		Flag for popular plans
plan_type	VARCHAR	ENUM(prepaid, postpaid)	Plan type
group_id	INT	FK → PlanGroups.group_id	Associated plan group identifier
description	TEXT		Plan description
criteria	JSON		Plan eligibility criteria
created_at	TIMESTAMP		Record creation timestamp
created_by	INT	FK → Admins.admin_id	Admin who created the plan

Column Name	Data Type	Constraints	Description
status	VARCHAR	ENUM(active, inactive)	Plan status

Offers Table

Purpose: Manages promotional offers for users.

Column Name	Data Type	Constraints	Description
offer_id	INT	PK	Unique offer identifier
offer_name	VARCHAR		Offer name
offer_validity	INT		Offer validity period (days)
offer_type_id	INT	FK → OfferTypes.offer_type_id	Associated offer type identifier
is_special	BOOLEAN		Flag for special offers
criteria	JSON		Offer eligibility criteria
description	TEXT		Offer description
created_at	TIMESTAMP		Record creation timestamp
created_by	INT	FK → Admins.admin_id	Admin who created the offer
status	VARCHAR	ENUM(active, inactive)	Offer status

CurrentActivePlans Table

Purpose: Tracks active plans assigned to users.

Column Name	Data Type	Constraints	Description
id	INT	PK	Unique record identifier
user_id	INT	FK → Users.user_id	Linked user identifier
plan_id	INT	FK → Plans.plan_id	Associated plan identifier
phone_number	VARCHAR		Associated phone number
valid_from	TIMESTAMP		Plan start timestamp
valid_to	TIMESTAMP		Plan expiry timestamp
status	VARCHAR	ENUM(active, expired, queued)	Plan status

Notifications Table

Purpose: Stores notification details for users and admins.

Column Name	Data Type	Constraints	Description
notification_id	INT	PK	Unique notification identifier
sender_type	VARCHAR	ENUM(system, admin)	Sender type
sender_id	INT	FK → Admins.admin_id	Admin sender identifier
description	TEXT		Notification content
recipient_type_id	INT	FK → NotificationRecipientTypes.recipient_type_id	Recipient type identifier
recipient_id	INT	FK → Users.user_id	Recipient user identifier
status	VARCHAR	ENUM(delivered, viewed)	Notification status
type_id	INT	FK → NotificationTypes.type_id	Notification type identifier
attachments	JSON		Notification attachments
timestamp	TIMESTAMP		Notification creation timestamp

ReferralRewards Table

Purpose: Records rewards for user referrals.

Column Name	Data Type	Constraints	Description
reward_id	INT	PK	Unique reward identifier
referrer_id	INT	FK → Users.user_id	Referrer user identifier
referred_id	INT	FK → Users.user_id	Referred user identifier
reward_amount	DECIMAL(10,2)		Reward amount
status	VARCHAR	ENUM(pending, earned, paid)	Reward status
created_at	TIMESTAMP		Record creation timestamp

AutoPay Table

Purpose: Manages automatic payment settings for users.

Column Name	Data Type	Constraints	Description
autopay_id	INT	PK	Unique autopay identifier
user_id	INT	FK → Users.user_id	Linked user identifier
plan_id	INT	FK → Plans.plan_id	Associated plan identifier
status	VARCHAR	ENUM(enabled, disabled)	Autopay status
tag	VARCHAR	ENUM(onetime, regular)	Autopay type
next_due_date	TIMESTAMP		Next payment due date
created_at	TIMESTAMP		Record creation timestamp

BillReminders Table

Purpose: Tracks billing reminders for users.

Column Name	Data Type	Constraints	Description
reminder_id	INT	PK	Unique reminder identifier
user_id	INT	FK → Users.user_id	Linked user identifier
plan_id	INT	FK → Plans.plan_id	Associated plan identifier
due_date	TIMESTAMP		Payment due date
status	VARCHAR	ENUM(pending, sent, paid)	Reminder status
sent_at	TIMESTAMP		Reminder sent timestamp

PlanGroups Table

Purpose: Defines categories for plans.

Column Name	Data Type	Constraints	Description
group_id	INT	PK	Unique plan group identifier
group_name	VARCHAR	UNIQUE	Plan group name (e.g., talktime, unlimited)

OfferTypes Table

Purpose: Defines types of promotional offers.

Column Name	Data Type	Constraints	Description
offer_type_id	INT	PK	Unique offer type identifier
offer_type_name	VARCHAR	UNIQUE	Offer type name (e.g., cashback, bonus_data)

Backup Table

Purpose: Manages database backup metadata.

Column Name	Data Type	Constraints	Description
backup_id	VARCHAR	PK	Unique backup identifier
backup_data	VARCHAR		Type of data backed up (e.g., users, orders)
snapshot_name	VARCHAR		Backup snapshot name (e.g., backup_2025_10_06_10_00)
source_db	VARCHAR		Source database (e.g., PostgreSQL)
storage_uri	VARCHAR		Backup storage location (e.g., S3 URI)
backup_status	VARCHAR	ENUM(failed, success)	Backup status
size_bytes	BIGINT		Backup size in bytes
description	VARCHAR		Backup description
details	JSON		Additional backup details
created_at	DATETIME		Record creation timestamp
created_by	INT	FK → Admins.admin_id	Admin who created the backup

4.2 MongoDB Collections

AuditLog Collection

Purpose: Tracks system events and user actions.

Field Name	Data Type	Constraints	Description
audit_id	VARCHAR	PK, ObjectId	Unique log entry identifier
user_id	VARCHAR	ObjectId	User performing the action
table_name	VARCHAR		Affected table/collection
action_type	VARCHAR	ENUM(INSERT, UPDATE, DELETE)	Type of action performed
details	JSON		Action details
created_at	DATETIME		Action timestamp

ContactForm Collection

Purpose: Stores user-submitted contact form data.

Field Name	Data Type	Constraints	Description
id	VARCHAR	PK, ObjectId	Unique contact form identifier
name	VARCHAR		Submitter's name
email	EMAIL		Submitter's email address
description	VARCHAR		Contact form message

BackupRestoreLogs Collection

Purpose: Logs backup and restore operations.

Field Name	Data Type	Constraints	Description
log_id	VARCHAR	PK	Unique log entry identifier
job_type	VARCHAR	ENUM(backup, restore, scheduled)	Type of job
job_id	VARCHAR	FK → Backup.backup_id	Associated backup identifier
status	VARCHAR	ENUM(started, progress, completed, failed)	Job status
details	JSON		Job details (e.g., tables affected)
created_at	DATETIME		Log creation timestamp
action_by	INT	FK → Admins.admin_id	Admin who performed the action

ReportsLogs Collection

Purpose: Stores report generation logs.

Field Name	Data Type	Constraints	Description
report_id	INT	PK	Unique report identifier
report_type_id	INT	FK → ReportTypes.report_type_id	Associated report type identifier
generated_by	INT	FK → Admins.admin_id	Admin who generated the report
format	VARCHAR	ENUM(pdf, excel, csv)	Report format
details	JSON		Report details

Field Name	Data Type	Constraints	Description
created_at	TIMESTAMP		Report creation timestamp

Content Collection

Purpose: Manages static and dynamic content (e.g., FAQs, policies).

Field Name	Data Type	Constraints	Description
content_id	INT	PK	Unique content identifier
content_type_id	INT	FK → ContentTypes.content_type_id	Associated content type identifier
title	VARCHAR		Content title
body	JSON		Content body
created_by	INT	FK → Admins.admin_id	Admin who created the content
updated_at	TIMESTAMP		Last update timestamp

5. RELATIONSHIPS AND CONSTRAINTS

5.1 PostgreSQL

Users Table

```
-- Users.role_id must reference Roles.role_id
ALTER TABLE Users
  ADD CONSTRAINT fk_users_role
  FOREIGN KEY (role_id) REFERENCES Roles(role_id);

-- Users.referred_by must reference Users.referral_code
ALTER TABLE Users
  ADD CONSTRAINT fk_users_referred_by
  FOREIGN KEY (referred_by) REFERENCES Users(referral_code);

-- Ensure email, phone_number, and referral_code are unique
ALTER TABLE Users
  ADD CONSTRAINT uq_users_email UNIQUE (email),
```

```
ADD CONSTRAINT uq_users_phone_number UNIQUE (phone_number),  
ADD CONSTRAINT uq_users_referral_code UNIQUE (referral_code);
```

UsersArchive Table

```
-- UsersArchive.role_id must reference Roles.role_id  
ALTER TABLE UsersArchive  
  ADD CONSTRAINT fk_usersarchive_role  
  FOREIGN KEY (role_id) REFERENCES Roles(role_id);  
  
-- UsersArchive.referred_by must reference Users.referral_code  
ALTER TABLE UsersArchive  
  ADD CONSTRAINT fk_usersarchive_referred_by  
  FOREIGN KEY (referred_by) REFERENCES Users(referral_code);  
  
-- Ensure email, phone_number, and referral_code are unique  
ALTER TABLE UsersArchive  
  ADD CONSTRAINT uq_usersarchive_email UNIQUE (email),  
  ADD CONSTRAINT uq_usersarchive_phone_number UNIQUE (phone_number),  
  ADD CONSTRAINT uq_usersarchive_referral_code UNIQUE (referral_code);
```

Sessions Table

```
-- Sessions.user_id must reference Users.user_id with cascade on delete  
ALTER TABLE Sessions  
  ADD CONSTRAINT fk_sessions_user  
  FOREIGN KEY (user_id) REFERENCES Users(user_id) ON DELETE CASCADE;  
  
-- Ensure refresh_token is unique and not null  
ALTER TABLE Sessions  
  ADD CONSTRAINT uq_sessions_refresh_token UNIQUE (refresh_token),  
  ALTER COLUMN refresh_token SET NOT NULL,  
  ALTER COLUMN refresh_token_expires_at SET NOT NULL;
```

Admins Table

```

-- Admins.role_id must reference Roles.role_id
ALTER TABLE Admins
  ADD CONSTRAINT fk_admins_role
  FOREIGN KEY (role_id) REFERENCES Roles(role_id);

-- Ensure email and phone_number are unique
ALTER TABLE Admins
  ADD CONSTRAINT uq_admins_email UNIQUE (email),
  ADD CONSTRAINT uq_admins_phone_number UNIQUE (phone_number);

```

RolePermissions Table

```

-- RolePermissions.role_id must reference Roles.role_id
ALTER TABLE RolePermissions
  ADD CONSTRAINT fk_rolepermissions_role
  FOREIGN KEY (role_id) REFERENCES Roles(role_id);

-- RolePermissions.permission_id must reference Permissions.permission_id
ALTER TABLE RolePermissions
  ADD CONSTRAINT fk_rolepermissions_permission
  FOREIGN KEY (permission_id) REFERENCES Permissions(permission_id);

```

Transactions Table

```

-- Transactions.user_id must reference Users.user_id
ALTER TABLE Transactions
  ADD CONSTRAINT fk_transactions_user
  FOREIGN KEY (user_id) REFERENCES Users(user_id);

-- Transactions.plan_id must reference Plans.plan_id
ALTER TABLE Transactions
  ADD CONSTRAINT fk_transactions_plan
  FOREIGN KEY (plan_id) REFERENCES Plans(plan_id);

-- Transactions.offer_id must reference Offers.offer_id
ALTER TABLE Transactions

```

```
ADD CONSTRAINT fk_transactions_offer
FOREIGN KEY (offer_id) REFERENCES Offers(offer_id);
```

Plans Table

```
-- Plans.group_id must reference PlanGroups.group_id
ALTER TABLE Plans
  ADD CONSTRAINT fk_plans_group
  FOREIGN KEY (group_id) REFERENCES PlanGroups(group_id);

-- Plans.created_by must reference Admins.admin_id
ALTER TABLE Plans
  ADD CONSTRAINT fk_plans_created_by
  FOREIGN KEY (created_by) REFERENCES Admins(admin_id);
```

Offers Table

```
-- Offers.offer_type_id must reference OfferTypes.offer_type_id
ALTER TABLE Offers
  ADD CONSTRAINT fk_offers_offer_type
  FOREIGN KEY (offer_type_id) REFERENCES OfferTypes(offer_type_id);

-- Offers.created_by must reference Admins.admin_id
ALTER TABLE Offers
  ADD CONSTRAINT fk_offers_created_by
  FOREIGN KEY (created_by) REFERENCES Admins(admin_id);
```

CurrentActivePlans Table

```
-- CurrentActivePlans.user_id must reference Users.user_id
ALTER TABLE CurrentActivePlans
  ADD CONSTRAINT fk_currentactiveplans_user
  FOREIGN KEY (user_id) REFERENCES Users(user_id);

-- CurrentActivePlans.plan_id must reference Plans.plan_id
ALTER TABLE CurrentActivePlans
```

```
ADD CONSTRAINT fk_currentactiveplans_plan  
FOREIGN KEY (plan_id) REFERENCES Plans(plan_id);
```

Notifications Table

```
-- Notifications.sender_id must reference Admins.admin_id  
ALTER TABLE Notifications  
  ADD CONSTRAINT fk_notifications_sender  
  FOREIGN KEY (sender_id) REFERENCES Admins(admin_id);  
  
-- Notifications.recipient_id must reference Users.user_id  
ALTER TABLE Notifications  
  ADD CONSTRAINT fk_notifications_recipient  
  FOREIGN KEY (recipient_id) REFERENCES Users(user_id);  
  
-- Notifications.recipient_type_id must reference NotificationRecipientTypes.recipient  
_type_id  
ALTER TABLE Notifications  
  ADD CONSTRAINT fk_notifications_recipient_type  
  FOREIGN KEY (recipient_type_id) REFERENCES NotificationRecipientTypes(recipient_type_id);  
  
-- Notifications.type_id must reference NotificationTypes.type_id  
ALTER TABLE Notifications  
  ADD CONSTRAINT fk_notifications_type  
  FOREIGN KEY (type_id) REFERENCES NotificationTypes(type_id);
```

ReferralRewards Table

```
-- ReferralRewards.referrer_id must reference Users.user_id  
ALTER TABLE ReferralRewards  
  ADD CONSTRAINT fk_referralrewards_referrer  
  FOREIGN KEY (referrer_id) REFERENCES Users(user_id);  
  
-- ReferralRewards.referred_id must reference Users.user_id  
ALTER TABLE ReferralRewards
```

```
ADD CONSTRAINT fk_referralrewards_referred  
FOREIGN KEY (referred_id) REFERENCES Users(user_id);
```

AutoPay Table

```
-- AutoPay.user_id must reference Users.user_id  
ALTER TABLE AutoPay  
  ADD CONSTRAINT fk_autopay_user  
  FOREIGN KEY (user_id) REFERENCES Users(user_id);  
  
-- AutoPay.plan_id must reference Plans.plan_id  
ALTER TABLE AutoPay  
  ADD CONSTRAINT fk_autopay_plan  
  FOREIGN KEY (plan_id) REFERENCES Plans(plan_id);
```

BillReminders Table

```
-- BillReminders.user_id must reference Users.user_id  
ALTER TABLE BillReminders  
  ADD CONSTRAINT fk_billreminders_user  
  FOREIGN KEY (user_id) REFERENCES Users(user_id);  
  
-- BillReminders.plan_id must reference Plans.plan_id  
ALTER TABLE BillReminders  
  ADD CONSTRAINT fk_billreminders_plan  
  FOREIGN KEY (plan_id) REFERENCES Plans(plan_id);
```

PlanGroups Table

```
-- Ensure group_name is unique  
ALTER TABLE PlanGroups  
  ADD CONSTRAINT uq_plangroups_group_name UNIQUE (group_name);
```

OfferTypes Table

```
-- Ensure offer_type_name is unique  
ALTER TABLE OfferTypes
```

```
ADD CONSTRAINT uq_offertypes_offer_type_name UNIQUE (offer_type_name);
```

Backup Table

```
-- Backup.created_by must reference Admins.admin_id
ALTER TABLE Backup
  ADD CONSTRAINT fk_backup_created_by
  FOREIGN KEY (created_by) REFERENCES Admins(admin_id);
```

Schedule Table

```
-- Schedule.backup_id must reference Backup.backup_id
ALTER TABLE Schedule
  ADD CONSTRAINT fk_schedule_backup
  FOREIGN KEY (backup_id) REFERENCES Backup(backup_id);
```

5.2 MongoDB

AuditLog Collection

```
// Validate AuditLog collection user_id is present and valid ObjectId
db.createCollection("AuditLog", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["user_id", "table_name", "action_type"],
      properties: {
        audit_id: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        user_id: {
          bsonType: "string",
          description: "must be a string (ObjectId) and is required"
        },
        table_name: {
          bsonType: "string",
```

```

        description: "must be a string and is required"
    },
    action_type: {
        enum: ["INSERT", "UPDATE", "DELETE"],
        description: "must be one of INSERT, UPDATE, DELETE"
    }
}
}
}
}
})

```

ContactForm Collection

```

// Validate ContactForm collection email is valid
db.createCollection("ContactForm", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["id", "email"],
      properties: {
        id: {
          bsonType: "string",
          description: "must be a string (ObjectId) and is required"
        },
        email: {
          bsonType: "string",
          pattern: "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$",
          description: "must be a valid email address"
        }
      }
    }
  }
})

```

BackupRestoreLogs Collection

```

// Validate BackupRestoreLogs collection job_id is present and job_type is valid
db.createCollection("BackupRestoreLogs", {

```

```

validator: {
  $jsonSchema: {
    bsonType: "object",
    required: ["job_id", "job_type", "status"],
    properties: {
      log_id: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      job_id: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      job_type: {
        enum: ["backup", "restore", "scheduled"],
        description: "must be one of backup, restore, scheduled"
      },
      status: {
        enum: ["started", "progress", "completed", "failed"],
        description: "must be one of started, progress, completed, failed"
      }
    }
  }
}
})

```

ReportsLogs Collection

```

// Validate ReportsLogs collection report_type_id and format are valid
db.createCollection("ReportsLogs", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["report_type_id", "format"],
      properties: {
        report_id: {
          bsonType: "int",
          description: "must be an integer and is required"
        }
      }
    }
  }
})

```

```

    },
    report_type_id: {
      bsonType: "int",
      description: "must be an integer and is required"
    },
    format: {
      enum: ["pdf", "excel", "csv"],
      description: "must be one of pdf, excel, csv"
    }
  }
}
}
})

```

Content Collection

```

// Validate Content collection content_type_id is present
db.createCollection("Content", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["content_type_id", "title"],
      properties: {
        content_id: {
          bsonType: "int",
          description: "must be an integer and is required"
        },
        content_type_id: {
          bsonType: "int",
          description: "must be an integer and is required"
        },
        title: {
          bsonType: "string",
          description: "must be a string and is required"
        }
      }
    }
  }
}

```

```
}  
})
```

6. VIEWS

6.1 PostgreSQL Views

User Permissions View

Purpose: Provides a quick lookup of user permissions by joining user, role, and permission data.

```
CREATE OR REPLACE VIEW user_permissions_v AS  
SELECT  
    u.user_id,  
    u.name AS user_name,  
    r.role_name,  
    p.resource AS permission_name,  
    p.read,  
    p.write,  
    p.delete,  
    p.edit  
FROM  
    Users u  
JOIN  
    Roles r ON u.role_id = r.role_id  
JOIN  
    RolePermissions rp ON r.role_id = rp.role_id  
JOIN  
    Permissions p ON rp.permission_id = p.permission_id;
```

Transaction Summary View

Purpose: Summarizes transaction details with user and plan information for reporting purposes.

```
CREATE OR REPLACE VIEW transaction_summary_v AS  
SELECT
```

```

t.txn_id,
t.user_id,
u.name AS user_name,
t.category,
t.txn_type,
t.amount,
t.service_type,
p.plan_name,
t.status AS transaction_status,
t.payment_method,
t.created_at
FROM
    Transactions t
JOIN
    Users u ON t.user_id = u.user_id
LEFT JOIN
    Plans p ON t.plan_id = p.plan_id
WHERE
    t.status IS NOT NULL;

```

Active Plans View

Purpose: Displays active plans with user and plan details for monitoring active subscriptions.

```

CREATE OR REPLACE VIEW active_plans_v AS
SELECT
    cap.id,
    cap.user_id,
    u.name AS user_name,
    cap.plan_id,
    p.plan_name,
    cap.phone_number,
    cap.valid_from,
    cap.valid_to,
    cap.status AS plan_status
FROM
    CurrentActivePlans cap
JOIN

```

```

    Users u ON cap.user_id = u.user_id
JOIN
    Plans p ON cap.plan_id = p.plan_id
WHERE
    cap.status = 'active';

```

6.2 MongoDB Views

Recent Audit Logs View

Purpose: Provides a view of recent audit logs with user action details for monitoring system activity.

```

db.createView(
  "recent_audit_logs_v",
  "AuditLog", // Source Collection
  [
    {
      $match: {
        created_at: { $gte: new Date(new Date().setDate(new Date().getDate() - 30)) } // Last 30 days
      }
    },
    { $sort: { created_at: -1 } },
    {
      $project: {
        _id: 0,
        audit_id: 1,
        user_id: 1,
        table_name: 1,
        action_type: 1,
        details: 1,
        created_at: 1
      }
    }
  ]
);

```

Content Summary View

Purpose: Summarizes content details with content type for easy access to FAQs, policies, etc.

```
db.createView(  
  "content_summary_v",  
  "Content", // Source Collection  
  [  
    {  
      $lookup: {  
        from: "ContentTypes",  
        localField: "content_type_id",  
        foreignField: "content_type_id",  
        as: "content_type_info"  
      }  
    },  
    { $unwind: "$content_type_info" },  
    {  
      $project: {  
        _id: 0,  
        content_id: 1,  
        title: 1,  
        content_type_name: "$content_type_info.content_type_name",  
        created_by: 1,  
        updated_at: 1  
      }  
    }  
  ]  
);
```

Recent Reports View

Purpose: Displays recent report logs with report type details for administrative review.

```
db.createView(  
  "recent_reports_v",  
  "ReportsLogs", // Source Collection  
  [  
    {  
      $lookup: {  
        from: "ReportTypes",  
        localField: "report_type_id",  
        foreignField: "report_type_id",  
        as: "report_type_info"  
      }  
    },  
    { $unwind: "$report_type_info" },  
    {  
      $project: {  
        _id: 0,  
        report_id: 1,  
        report_type_name: "$report_type_info.report_type_name",  
        created_at: 1,  
        updated_at: 1  
      }  
    }  
  ]  
);
```

```

{
  $match: {
    created_at: { $gte: new Date(new Date().setDate(new Date().getDate() - 3
0)) } // Last 30 days
  }
},
{
  $lookup: {
    from: "ReportTypes",
    localField: "report_type_id",
    foreignField: "report_type_id",
    as: "report_type_info"
  }
},
{ $unwind: "$report_type_info" },
{ $sort: { created_at: -1 } },
{
  $project: {
    _id: 0,
    report_id: 1,
    report_type_name: "$report_type_info.report_type_name",
    format: 1,
    generated_by: 1,
    created_at: 1
  }
}
]
);

```

7. PROCEDURES/FUNCTIONS

7.1 PostgreSQL Functions

Set Transaction Timestamp Function

Purpose: Automatically sets the `created_at` timestamp for new transactions to the current time.

```
CREATE OR REPLACE FUNCTION set_transaction_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.created_at = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger to apply the function on insert
CREATE TRIGGER trigger_set_transaction_timestamp
BEFORE INSERT ON Transactions
FOR EACH ROW
EXECUTE FUNCTION set_transaction_timestamp();
```

Update Wallet Balance Function

Purpose: Updates the `wallet_balance` in the `Users` table based on a transaction's `amount` and `txn_type`, ensuring consistency.

```
CREATE OR REPLACE FUNCTION update_wallet_balance()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.txn_type = 'credit' THEN
        UPDATE Users
        SET wallet_balance = wallet_balance + NEW.amount
        WHERE user_id = NEW.user_id;
    ELSIF NEW.txn_type = 'debit' THEN
        UPDATE Users
        SET wallet_balance = wallet_balance - NEW.amount
        WHERE user_id = NEW.user_id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger to apply the function after insert
```

```
CREATE TRIGGER trigger_update_wallet_balance
AFTER INSERT ON Transactions
FOR EACH ROW
EXECUTE FUNCTION update_wallet_balance();
```

Check Plan Expiry Function

Purpose: Marks plans in `CurrentActivePlans` as expired if the `valid_to` date is in the past.

```
CREATE OR REPLACE FUNCTION check_plan_expiry()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.valid_to < NOW() THEN
        NEW.status = 'expired';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Trigger to apply the function on insert or update
CREATE TRIGGER trigger_check_plan_expiry
BEFORE INSERT OR UPDATE ON CurrentActivePlans
FOR EACH ROW
EXECUTE FUNCTION check_plan_expiry();
```

7.2 MongoDB Aggregation Pipelines

User Activity Summary Pipeline

Purpose: Aggregates user activity from `AuditLog` to summarize the number of actions per user and action type over the last 30 days.

```
db.AuditLog.aggregate([
  {
    $match: {
      created_at: { $gte: new Date(new Date().setDate(new Date().getDate() - 30)) }
    }
  },
  {
    $group: {
      _id: '$user_id',
      actions: { $sum: '$count' }
    }
  }
])
```

```

    $group: {
      _id: {
        user_id: "$user_id",
        action_type: "$action_type"
      },
      total_actions: { $sum: 1 }
    },
    {
      $project: {
        _id: 0,
        user_id: "$_id.user_id",
        action_type: "$_id.action_type",
        total_actions: 1
      }
    },
    { $sort: { user_id: 1, action_type: 1 } }
  ]);

```

Content Usage Report Pipeline

Purpose: Summarizes content access by type, counting the number of content items per

`content_type_name` .

```

db.Content.aggregate([
  {
    $lookup: {
      from: "ContentTypes",
      localField: "content_type_id",
      foreignField: "content_type_id",
      as: "content_type_info"
    }
  },
  {
    $unwind: "$content_type_info"
  },
  {
    $group: {
      _id: "$content_type_info.content_type_name",

```

```

    total_items: { $sum: 1 }
  }
},
{
  $project: {
    _id: 0,
    content_type_name: "$_id",
    total_items: 1
  }
},
{ $sort: { content_type_name: 1 } }
]);

```

Recent Backup Restore Activity Pipeline

Purpose: Lists recent backup and restore activities from `BackupRestoreLogs`, including job type and status, for the last 30 days.

```

db.BackupRestoreLogs.aggregate([
  {
    $match: {
      created_at: { $gte: new Date(new Date().setDate(new Date().getDate() - 30)) }
    }
  },
  {
    $lookup: {
      from: "Backup",
      localField: "job_id",
      foreignField: "backup_id",
      as: "backup_info"
    }
  },
  {
    $unwind: { path: "$backup_info", preserveNullAndEmptyArrays: true }
  },
  {
    $project: {
      _id: 0,
      log_id: 1,

```

```

    job_type: 1,
    status: 1,
    snapshot_name: "$backup_info.snapshot_name",
    created_at: 1,
    action_by: 1
  }
},
{ $sort: { created_at: -1 } }
]);

```

8. TRIGGERS

8.1 PostgreSQL Triggers

Transaction Timestamp Trigger

Purpose: Automatically sets the `created_at` timestamp for new records in the `Transactions` table to the current time.

```

CREATE OR REPLACE FUNCTION set_transaction_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.created_at = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER transaction_timestamp_trigger
BEFORE INSERT ON Transactions
FOR EACH ROW
EXECUTE FUNCTION set_transaction_timestamp();

```

Wallet Balance Update Trigger

Purpose: Updates the `wallet_balance` in the `Users` table when a new transaction is inserted, based on the transaction's `amount` and `txn_type`.

```

CREATE OR REPLACE FUNCTION update_wallet_balance()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.txn_type = 'credit' THEN
        UPDATE Users
        SET wallet_balance = wallet_balance + NEW.amount
        WHERE user_id = NEW.user_id;
    ELSIF NEW.txn_type = 'debit' THEN
        IF (SELECT wallet_balance FROM Users WHERE user_id = NEW.user_id) >= NE
W.amount THEN
            UPDATE Users
            SET wallet_balance = wallet_balance - NEW.amount
            WHERE user_id = NEW.user_id;
        ELSE
            RAISE EXCEPTION 'Insufficient wallet balance for user_id %', NEW.user_id;
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER wallet_balance_trigger
AFTER INSERT ON Transactions
FOR EACH ROW
EXECUTE FUNCTION update_wallet_balance();

```

Plan Expiry Status Trigger

Purpose: Automatically updates the `status` in the `CurrentActivePlans` table to 'expired' if the `valid_to` date is in the past during insert or update.

```

CREATE OR REPLACE FUNCTION check_plan_expiry()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.valid_to < NOW() THEN
        NEW.status = 'expired';
    END IF;
    RETURN NEW;
END;

```

```

END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER plan_expiry_trigger
BEFORE INSERT OR UPDATE ON CurrentActivePlans
FOR EACH ROW
EXECUTE FUNCTION check_plan_expiry();

```

Notification Timestamp Trigger

Purpose: Automatically sets the `timestamp` for new records in the `Notifications` table to the current time.

```

CREATE OR REPLACE FUNCTION set_notification_timestamp()
RETURNS TRIGGER AS $$
BEGIN
    NEW.timestamp = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER notification_timestamp_trigger
BEFORE INSERT ON Notifications
FOR EACH ROW
EXECUTE FUNCTION set_notification_timestamp();

```

Referral Reward Status Trigger

Purpose: Updates the `status` in the `ReferralRewards` table to 'earned' when a new reward is inserted and the referred user is active.

```

CREATE OR REPLACE FUNCTION set_referral_reward_status()
RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (SELECT 1 FROM Users WHERE user_id = NEW.referred_id AND status =
'active') THEN
        NEW.status = 'earned';
    ELSE
        NEW.status = 'pending';

```

```

    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER referral_reward_status_trigger
BEFORE INSERT ON ReferralRewards
FOR EACH ROW
EXECUTE FUNCTION set_referral_reward_status();

```

9. INDEX SPECIFICATIONS

9.1 PostgreSQL Index Specifications

Users Table Indexes

```

-- To quickly find user details by email or phone number
CREATE INDEX idx_users_email ON Users(email);
CREATE INDEX idx_users_phone_number ON Users(phone_number);
-- To efficiently query users by referral code
CREATE INDEX idx_users_referral_code ON Users(referral_code);
-- To efficiently query users by role
CREATE INDEX idx_users_role_id ON Users(role_id);

```

UsersArchive Table Indexes

```

-- To quickly find archived user details by email or phone number
CREATE INDEX idx_usersarchive_email ON UsersArchive(email);
CREATE INDEX idx_usersarchive_phone_number ON UsersArchive(phone_number);
-- To efficiently query archived users by referral code
CREATE INDEX idx_usersarchive_referral_code ON UsersArchive(referral_code);

```

Sessions Table Indexes

```
-- To quickly find and validate a session by refresh token
CREATE INDEX idx_sessions_refresh_token ON Sessions(refresh_token);
-- To find all active sessions for a given user
CREATE INDEX idx_sessions_user_id ON Sessions(user_id);
-- To periodically clean up expired sessions
CREATE INDEX idx_sessions_refresh_token_expires_at ON Sessions(refresh_token_expires_at);
```

Admins Table Indexes

```
-- To quickly find admin details by email
CREATE INDEX idx_admins_email ON Admins(email);
-- To efficiently query admins by role
CREATE INDEX idx_admins_role_id ON Admins(role_id);
```

RolePermissions Table Indexes

```
-- To quickly find all permissions associated with a role
CREATE INDEX idx_rolepermissions_role_id ON RolePermissions(role_id);
-- To efficiently query permissions by permission_id
CREATE INDEX idx_rolepermissions_permission_id ON RolePermissions(permission_id);
```

Transactions Table Indexes

```
-- To find all transactions for a specific user
CREATE INDEX idx_transactions_user_id ON Transactions(user_id);
-- To efficiently query transactions by plan or offer
CREATE INDEX idx_transactions_plan_id ON Transactions(plan_id);
CREATE INDEX idx_transactions_offer_id ON Transactions(offer_id);
-- To query transactions by date range
CREATE INDEX idx_transactions_created_at ON Transactions(created_at DESC);
```

Plans Table Indexes

```
-- To efficiently query plans by group
CREATE INDEX idx_plans_group_id ON Plans(group_id);
-- To filter plans by status or type
CREATE INDEX idx_plans_status ON Plans(status);
CREATE INDEX idx_plans_plan_type ON Plans(plan_type);
```

Offers Table Indexes

```
-- To efficiently query offers by offer type
CREATE INDEX idx_offers_offer_type_id ON Offers(offer_type_id);
-- To filter offers by status
CREATE INDEX idx_offers_status ON Offers(status);
```

CurrentActivePlans Table Indexes

```
-- To find all active plans for a specific user
CREATE INDEX idx_currentactiveplans_user_id ON CurrentActivePlans(user_id);
-- To efficiently query active plans by plan
CREATE INDEX idx_currentactiveplans_plan_id ON CurrentActivePlans(plan_id);
-- To query plans by status or validity
CREATE INDEX idx_currentactiveplans_status ON CurrentActivePlans(status);
CREATE INDEX idx_currentactiveplans_valid_to ON CurrentActivePlans(valid_to);
```

Notifications Table Indexes

```
-- To find notifications for a specific user or recipient type
CREATE INDEX idx_notifications_recipient_id ON Notifications(recipient_id);
CREATE INDEX idx_notifications_recipient_type_id ON Notifications(recipient_type_id);
-- To query notifications by timestamp
CREATE INDEX idx_notifications_timestamp ON Notifications(timestamp DESC);
```

ReferralRewards Table Indexes

```
-- To find rewards for a specific referrer or referred user
CREATE INDEX idx_referralrewards_referrer_id ON ReferralRewards(referrer_id);
```

```
CREATE INDEX idx_referralrewards_referred_id ON ReferralRewards(referred_id);  
-- To filter rewards by status  
CREATE INDEX idx_referralrewards_status ON ReferralRewards(status);
```

Backup Table Indexes

```
-- To efficiently query backups by status or creation date  
CREATE INDEX idx_backup_backup_status ON Backup(backup_status);  
CREATE INDEX idx_backup_created_at ON Backup(created_at DESC);
```

Schedule Table Indexes

```
-- To quickly find schedules by backup_id  
CREATE INDEX idx_schedule_backup_id ON Schedule(backup_id);  
-- To query schedules by next run date  
CREATE INDEX idx_schedule_next_run ON Schedule(next_run);
```

AutoPay Table Indexes

```
-- To find autopay settings for a specific user  
CREATE INDEX idx_autopay_user_id ON AutoPay(user_id);  
-- To query autopay by plan or status  
CREATE INDEX idx_autopay_plan_id ON AutoPay(plan_id);  
CREATE INDEX idx_autopay_status ON AutoPay(status);
```

BillReminders Table Indexes

```
-- To find reminders for a specific user  
CREATE INDEX idx_billreminders_user_id ON BillReminders(user_id);  
-- To query reminders by due date or status  
CREATE INDEX idx_billreminders_due_date ON BillReminders(due_date);  
CREATE INDEX idx_billreminders_status ON BillReminders(status);
```

9.2 MongoDB Index Specifications

AuditLog Collection Indexes

```
// To find all actions performed by a specific user
db.AuditLog.createIndex({ "user_id": 1 });
// To query audit events by time (descending for recent logs)
db.AuditLog.createIndex({ "created_at": -1 });
// To filter audit logs by action type
db.AuditLog.createIndex({ "action_type": 1 });
```

ContactForm Collection Indexes

```
// To quickly find contact forms by email
db.ContactForm.createIndex({ "email": 1 }, { unique: true });
// To query contact forms by submission time
db.ContactForm.createIndex({ "created_at": -1 });
```

BackupRestoreLogs Collection Indexes

```
// To find logs for a specific backup job
db.BackupRestoreLogs.createIndex({ "job_id": 1 });
// To query logs by creation time (descending for recent logs)
db.BackupRestoreLogs.createIndex({ "created_at": -1 });
// To filter logs by job type or status
db.BackupRestoreLogs.createIndex({ "job_type": 1 });
db.BackupRestoreLogs.createIndex({ "status": 1 });
```

ReportsLogs Collection Indexes

```
// To find reports by report type
db.ReportsLogs.createIndex({ "report_type_id": 1 });
// To query reports by generation time (descending for recent reports)
db.ReportsLogs.createIndex({ "created_at": -1 });
// To filter reports by format
db.ReportsLogs.createIndex({ "format": 1 });
```

Content Collection Indexes

```
// To find content by content type
db.Content.createIndex({ "content_type_id": 1 });
// To query content by update time
db.Content.createIndex({ "updated_at": -1 });
// To search content by title
db.Content.createIndex({ "title": 1 });
```

10. SECURITY CONSIDERATIONS

10.1 User Roles and Access Control

- **Admin:** Full read/write access to all data across MongoDB and PostgreSQL, including user management, transactions, plans, offers, notifications, and backup/restore operations. Admins can manage `Users`, `Admins`, `Plans`, `Offers`, `Content`, and `ReportsLogs` tables/collections, and perform actions like creating backups (`Backup`) and scheduling tasks (`Schedule`).
- **User:** Read/write access to their own profile (`Users`), transactions (`Transactions`), active plans (`CurrentActivePlans`), referral rewards (`ReferralRewards`), and notifications (`Notifications`) where `recipient_id` matches `user_id`). Users can view but not modify system-generated content in `Content` (e.g., FAQs, terms of service).
- **Authorization:** Implement role-based access control (RBAC) using the `Roles`, `Permissions`, and `RolePermissions` tables in PostgreSQL. Backend APIs must enforce permissions based on `role_id` in `Users` and `Admins`, ensuring users and admins only access authorized resources (e.g., `resource` in `Permissions`).

10.2 Data Encryption and Security

- **Sensitive Data Encryption:** Encrypt personally identifiable information (PII) fields such as `email`, `phone_number`, and `name` in `Users`, `UsersArchive`, `Admins`, and `ContactForm` at rest using database-level encryption (e.g., PostgreSQL pgcrypto or MongoDB field-level encryption) or application-layer encryption.
- **Connection Security:** Use SSL/TLS encryption for all database connections to PostgreSQL and MongoDB to protect data in transit.
- **Session Security:** Securely store and manage `refresh_token` in the `Sessions` table using strong encryption. Use HttpOnly, Secure cookies or Authorization headers for session tokens in application logic.

- **Row-Level Security (RLS):** Enable RLS in PostgreSQL for tables like `Users` , `Transactions` , and `Notifications` to restrict access to rows based on `user_id` or `role_id` . For example, ensure users can only access their own records in `Transactions` (`user_id = current_user_id`).
- **MongoDB Access Controls:** Utilize MongoDB's built-in role-based access controls to restrict access to collections (`AuditLog` , `ReportsLogs` , etc.) and enable auditing for administrative actions.
- **Data Integrity:** Ensure `wallet_balance` in `Users` is updated securely via triggers (e.g., for `Transactions`) to prevent unauthorized modifications.

10.3 Backup and Disaster Recovery Strategy

- **Backup Schedule:** Schedule daily full backups of PostgreSQL (`Users` , `Transactions` , `Plans` , etc.) and MongoDB (`AuditLog` , `ReportsLogs` , etc.) databases at 2:00 AM during low-usage hours, recorded in the `Backup` table.
- **Incremental Backups:** Implement hourly incremental backups for PostgreSQL and MongoDB during business hours to capture changes with minimal performance impact, tracked in `BackupRestoreLogs` .
- **Retention Policy:** Retain backups for a rolling window of 30 days, as specified in `Backup.snapshot_name` , to enable point-in-time recovery.
- **Storage:** Store backups in secure, off-site cloud storage (e.g., AWS S3, as indicated by `Backup.storage_uri`) to ensure data durability and resilience.
- **Testing:** Regularly test restore procedures using `BackupRestoreLogs` to verify backup integrity and recovery readiness, ensuring `backup_status` is marked as 'success' or 'failed' appropriately.

10.4 Additional Security Best Practices

- **Audit Logging:** Maintain detailed audit logs in the `AuditLog` collection for user actions (e.g., `INSERT` , `UPDATE` , `DELETE` on tables like `Users` , `Transactions`), including `user_id` , `table_name` , and `action_type` . Log administrative changes in `Admins` and `Backup` tables.
- **Multi-Factor Authentication (MFA):** Implement MFA for admin logins (`Admins` table) to enhance access security, especially for actions involving `Plans` , `Offers` , and `Backup` management.
- **Network Security:** Apply IP whitelisting and firewalls to restrict database access to authorized application servers only. Limit access to PostgreSQL and MongoDB to specific IP ranges.

- **Patch Management:** Regularly update and patch PostgreSQL and MongoDB versions, along with application dependencies, to address security vulnerabilities.
- **Environment Hardening:** Secure database servers with hardened OS configurations, applying least privilege principles to database users and roles.
- **Session Management:** Periodically clean up expired sessions in the `Sessions` table (`refresh_token_expires_at < NOW()`) and revoke inactive sessions (`is_active = false`) to prevent unauthorized access.

11. PERFORMANCE OPTIMIZATION

11.1 Query Optimization Tips

- **Prepared Statements:** Use prepared statements in PostgreSQL for queries involving `Users` , `Transactions` , `Notifications` , and `Sessions` to prevent SQL injection and improve execution plan caching. For MongoDB, use parameterized queries in application code for collections like `AuditLog` and `ReportsLogs` to enhance security and performance.
- **Connection Pooling:** Implement connection pooling for both PostgreSQL and MongoDB to manage database connections efficiently, reducing overhead for high-frequency operations like user authentication (`Sessions`) and transaction processing (`Transactions`).
- **Caching Frequently Accessed Data:** Cache static or semi-static data, such as `Plans` , `Offers` , `PlanGroups` , `OfferTypes` , `NotificationTypes` , and `ContentTypes` , using an in-memory cache (e.g., Redis) to reduce database load for frequently accessed lookups, such as plan details or content (e.g., FAQs, terms of service).
- **Partitioning Large Tables:** Partition large tables like `Transactions` and `Notifications` by date ranges (e.g., monthly or yearly partitions on `created_at` or `timestamp`) to improve query performance and manage data growth. For example, partition `Transactions` by `created_at` to optimize historical transaction queries.
- **Regular Index Maintenance and Statistics Updates:** Regularly rebuild indexes on high-write tables like `Transactions` , `CurrentActivePlans` , and `Notifications` to maintain performance. Use `ANALYZE` in PostgreSQL to update table statistics for the query planner, ensuring optimal execution plans. For MongoDB, periodically review and optimize indexes on collections like `AuditLog` and `BackupRestoreLogs` using `collMod` or `createIndex` .

- **Denormalization for MongoDB:** For collections like `Content` and `ReportsLogs`, consider embedding frequently accessed fields (e.g., `content_type_name` from `ContentTypes`) to reduce the need for `$lookup` operations, improving query performance for common reports or content retrieval.
- **Query Aggregation Optimization:** In MongoDB, optimize aggregation pipelines for `AuditLog` and `ReportsLogs` by limiting the result set early (e.g., using `$match` on `created_at`) and indexing fields used in `$group` or `$sort` stages (e.g., `user_id`, `report_type_id`).

11.2 Monitoring

- **Track Slow Queries:** Monitor queries in PostgreSQL taking longer than 2 seconds using `pg_stat_statements` or logging, focusing on tables like `Transactions`, `Notifications`, and `CurrentActivePlans`. For MongoDB, use the query profiler to identify slow queries (> 2 seconds) in collections like `AuditLog` and `BackupRestoreLogs`.
- **Monitor Table/Collection Sizes and Growth Rates:** Track the size and growth of large tables like `Transactions`, `Notifications`, and `Sessions`, and collections like `AuditLog` and `ReportsLogs`. Use PostgreSQL's `pg_table_size` and MongoDB's `collStats` to monitor storage usage and plan for partitioning or sharding if needed.
- **Alert on Failed Backup Jobs:** Set up alerts for failed backup jobs by monitoring the `Backup.backup_status` field (for 'failed' status) and `BackupRestoreLogs.status` in MongoDB. Ensure notifications are triggered for any backup or restore job marked as 'failed' to address issues promptly.
- **Track Concurrent Connection Counts:** Monitor concurrent connections to PostgreSQL using `pg_stat_activity` to ensure the connection pool is not exhausted, especially during peak transaction or notification processing. For MongoDB, use `serverStatus` to track connection metrics and avoid overload on collections like `AuditLog` or `ReportsLogs`.
- **Monitor Index Usage:** Use PostgreSQL's `pg_stat_user_indexes` to track index usage on tables like `Users`, `Transactions`, and `CurrentActivePlans`, dropping unused indexes to reduce overhead. In MongoDB, use `explain()` to verify index usage for queries on `AuditLog` and `Content`.
- **Track Transaction and Notification Latency:** Monitor the latency of transaction processing (`Transactions.created_at`) and notification delivery (`Notifications.timestamp`) to identify bottlenecks in real-time operations, such as recharge or autopay transactions.

12. DATA MIGRATION

12.1 Initial Data Load Sequence

To ensure data integrity and respect foreign key constraints and dependencies, the initial data load for the telecommunications and user management system should follow the sequence below. The sequence is designed to populate foundational tables/collections first, followed by dependent tables/collections, ensuring that referenced data exists before inserting records that depend on it.

1. Roles (PostgreSQL)

- Load the `Roles` table to define user and admin roles (e.g., `role_id` , `role_name`).
- Reason: Required by `Users` , `UsersArchive` , and `Admins` for `role_id` references.

2. Permissions (PostgreSQL)

- Load the `Permissions` table to define access control resources (e.g., `permission_id` , `resource`).
- Reason: Required by `RolePermissions` for `permission_id` references.

3. RolePermissions (PostgreSQL)

- Load the `RolePermissions` table to map roles to permissions (e.g., `role_id` , `permission_id`).
- Reason: Depends on `Roles` and `Permissions` .

4. PlanGroups (PostgreSQL)

- Load the `PlanGroups` table to define plan categories (e.g., `group_id` , `group_name`).
- Reason: Required by `Plans` for `group_id` references.

5. OfferTypes (PostgreSQL)

- Load the `OfferTypes` table to define offer categories (e.g., `offer_type_id` , `offer_type_name`).
- Reason: Required by `Offers` for `offer_type_id` references.

6. Admins (PostgreSQL)

- Load the `Admins` table to define administrative users (e.g., `admin_id` , `email` , `role_id`).
- Reason: Depends on `Roles` for `role_id` ; required by `Plans` , `Offers` , `Content` , `Backup` , `BackupRestoreLogs` , and `ReportsLogs` for `created_by` or `action_by` references.

7. Users (PostgreSQL)

- Load the **Users** table to define customer profiles (e.g., **user_id**, **email**, **phone_number**, **role_id**).
- Reason: Depends on **Roles** for **role_id** and **Users.referral_code** for **referred_by** ; required by **Sessions**, **Transactions**, **CurrentActivePlans**, **Notifications**, **ReferralRewards**, and **AutoPay** .

8. UsersArchive (PostgreSQL)

- Load the **UsersArchive** table to store archived user data (e.g., **user_id**, **email**, **phone_number**).
- Reason: Depends on **Roles** and **Users.referral_code** for **referred_by** . Can be loaded after **Users** as it is an archive table.

9. Plans (PostgreSQL)

- Load the **Plans** table to define service plans (e.g., **plan_id**, **plan_name**, **group_id**, **created_by**).
- Reason: Depends on **PlanGroups** and **Admins** ; required by **Transactions**, **CurrentActivePlans**, **AutoPay**, and **BillReminders** .

10. Offers (PostgreSQL)

- Load the **Offers** table to define promotional offers (e.g., **offer_id**, **offer_name**, **offer_type_id**, **created_by**).
- Reason: Depends on **OfferTypes** and **Admins** ; required by **Transactions** .

11. Content (MongoDB)

- Load the **Content** collection to store system content (e.g., **content_id**, **title**, **content_type_id**, **created_by**).
- Reason: Depends on **ContentTypes** and **Admins** .

12. Sessions (PostgreSQL)

- Load the **Sessions** table to store user session data (e.g., **session_id**, **user_id**, **refresh_token**).
- Reason: Depends on **Users** for **user_id** .

13. Transactions (PostgreSQL)

- Load the **Transactions** table to store user transactions (e.g., **txn_id**, **user_id**, **plan_id**, **offer_id**).
- Reason: Depends on **Users**, **Plans**, and **Offers** .

14. CurrentActivePlans (PostgreSQL)

- Load the `CurrentActivePlans` table to track active user plans (e.g., `id`, `user_id`, `plan_id`).
- Reason: Depends on `Users` and `Plans`.

15. ReferralRewards (PostgreSQL)

- Load the `ReferralRewards` table to store referral rewards (e.g., `reward_id`, `referrer_id`, `referred_id`).
- Reason: Depends on `Users` for `referrer_id` and `referred_id`.

16. Notifications (PostgreSQL)

- Load the `Notifications` table to store user notifications (e.g., `notification_id`, `recipient_id`, `type_id`).
- Reason: Depends on `Users`, `Admins`, `NotificationRecipientTypes`, and `NotificationTypes`.

17. AutoPay (PostgreSQL)

- Load the `AutoPay` table to store automatic payment settings (e.g., `autopay_id`, `user_id`, `plan_id`).
- Reason: Depends on `Users` and `Plans`.

18. BillReminders (PostgreSQL)

- Load the `BillReminders` table to store billing reminders (e.g., `reminder_id`, `user_id`, `plan_id`).
- Reason: Depends on `Users` and `Plans`.

19. Backup (PostgreSQL)

- Load the `Backup` table to store backup metadata (e.g., `backup_id`, `snapshot_name`, `created_by`).
- Reason: Depends on `Admins`; required by `Schedule` and `BackupRestoreLogs`.

20. BackupRestoreLogs (MongoDB)

- Load the `BackupRestoreLogs` collection to store backup/restore activity logs (e.g., `log_id`, `job_id`, `action_by`).
- Reason: Depends on `Backup` and `Admins`.

21. ReportsLogs (MongoDB)

- Load the `ReportsLogs` collection to store report generation logs (e.g., `report_id`, `report_type_id`, `generated_by`).
- Reason: Depends on `ReportTypes` and `Admins`.

22. AuditLog (MongoDB)

- Load the `AuditLog` collection to store user action logs (e.g., `audit_id`, `user_id`, `action_type`).
- Reason: Depends on `Users` for `user_id`.

23. ContactForm (MongoDB)

- Load the `ContactForm` collection to store customer inquiries (e.g., `id`, `email`, `description`).
- Reason: No dependencies, can be loaded last as it is independent.

13. APPENDIX

13.1 Sample Roles Data (PostgreSQL)

```
INSERT INTO Roles (role_id, role_name) VALUES
(1, 'Admin', 'System administrator with full access to all data and operations'),
(2, 'User', 'Customer with access to their own profile, transactions, and plans'),
(3, 'Support', 'Support staff with limited access to user data and reports');
```

13.2 Sample Permissions Data (PostgreSQL)

```
INSERT INTO Permissions (permission_id, resource, read, write, delete, edit) VALUES
(1, 'Users', true, true, true, true),
(2, 'Transactions', true, false, false, false),
(3, 'Plans', true, true, false, true),
(4, 'Reports', true, true, true, true);
```

13.3 Sample RolePermissions Data (PostgreSQL)

```
INSERT INTO RolePermissions (id, role_id, permission_id) VALUES
(1, 1, 1), -- Admin: Full access to Users
(2, 1, 4), -- Admin: Full access to Reports
(3, 2, 2), -- User: Read-only access to Transactions
(4, 3, 2); -- Support: Read-only access to Transactions
```

13.4 Sample PlanGroups Data (PostgreSQL)

```
INSERT INTO PlanGroups (group_id, group_name) VALUES
(1, 'talktime', 'Plans offering talktime minutes'),
(2, 'unlimited', 'Plans with unlimited calls and data'),
(3, 'roaming', 'Plans for roaming services'),
(4, 'other', 'Miscellaneous plans');
```

13.5 Sample OfferTypes Data (PostgreSQL)

```
INSERT INTO OfferTypes (offer_type_id, offer_type_name) VALUES
(1, 'cashback', 'Offers providing cashback on recharges'),
(2, 'bonus_data', 'Offers providing additional data'),
(3, 'festive_special', 'Seasonal or festive promotional offers');
```

13.10 Sample Backup Data (PostgreSQL)

```
INSERT INTO Backup (backup_id, backup_data, snapshot_name, source_db, storage_
uri, backup_status, size_bytes, description, details, created_at, created_by) VALUES
('backup_2025_10_10_0200', 'users', 'backup_2025_10_10_0200', 'PostgreSQL', 's3://te
lecom-backups/2025_10_10', 'success', 104857600, 'Daily user data backup', '{"table
s": ["Users", "UsersArchive"]}', '2025-10-10 02:00:00', 1),
('backup_2025_10_11_0200', 'transactions', 'backup_2025_10_11_0200', 'PostgreSQL',
's3://telecom-backups/2025_10_11', 'success', 209715200, 'Daily transaction data bac
kup', '{"tables": ["Transactions"]}', '2025-10-11 02:00:00', 1);
```

13.11 Sample Schedule Data (PostgreSQL)

```
INSERT INTO Schedule (schedule_id, backup_id, task_name, time, next_run, isactive,
created_at) VALUES
('sched_001', 'backup_2025_10_10_0200', 'daily_backup', '2025-10-10', '2025-10-11',
'active', '2025-10-09 10:00:00'),
('sched_002', 'backup_2025_10_11_0200', 'weekly_backup', '2025-10-11', '2025-10-1
8', 'active', '2025-10-09 10:00:00');
```

13.12 Sample BackupRestoreLogs Data (MongoDB)

```
db.BackupRestoreLogs.insertMany([
  {
    log_id: "log_2025_10_10_0200",
    job_type: "backup",
    job_id: "backup_2025_10_10_0200",
    status: "completed",
    details: { "tables": ["Users"], "notes": "Full backup completed successfully" },
    created_at: ISODate("2025-10-10T02:00:00Z"),
    action_by: 1
  },
  {
    log_id: "log_2025_10_11_0300",
    job_type: "restore",
    job_id: "backup_2025_10_10_0200",
    status: "failed",
    details: { "tables": ["Users"], "notes": "Restore failed due to incomplete snapshot"
  },
  created_at: ISODate("2025-10-11T03:00:00Z"),
  action_by: 1
}
]);
```

13.13 Sample Content Data (MongoDB)

```
db.Content.insertMany([
  {
    content_id: 1,
    content_type_id: 1,
    title: "How to Recharge Your Plan",
    body: { "content": "Step-by-step guide to recharge your prepaid or postpaid pla
n..." },
    created_by: 1,
    updated_at: ISODate("2025-10-10T10:00:00Z")
  },
  {
```

```

    content_id: 2,
    content_type_id: 3,
    title: "Terms of Service",
    body: { "content": "Terms and conditions for using our telecom services..." },
    created_by: 1,
    updated_at: ISODate("2025-10-10T10:00:00Z")
  }
];

```

13.14 Sample Users Data (PostgreSQL)

```

INSERT INTO Users (user_id, name, email, phone_number, referral_code, referred_by,
user_type, status, wallet_balance, created_at, updated_at, role_id) VALUES
(1, 'Saravanan', 'saravanan@example.com', '9876543210', 'REF123', NULL, 'prepaid',
'active', 100.50, '2025-10-10 08:00:00', '2025-10-10 08:00:00', 2),
(2, 'Vijay', 'vijay@example.com', '8765432109', 'REF456', 'REF123', 'postpaid', 'active',
50.00, '2025-10-10 09:00:00', '2025-10-10 09:00:00', 2);

```

13.15 Sample Admins Data (PostgreSQL)

```

INSERT INTO Admins (admin_id, name, email, phone_number, status, created_at, updated_at, role_id) VALUES
(1, 'Admin User', 'admin@example.com', '9999999999', 'active', '2025-10-10 07:00:00', '2025-10-10 07:00:00', 1),
(2, 'Support User', 'support@example.com', '8888888888', 'active', '2025-10-10 07:30:00', '2025-10-10 07:30:00', 3);

```

14. DOCUMENT VERSION HISTORY

VERSION	DATE	AUTHOR	DESCRIPTION
2.0	10-11-2025	Saravanan B S	Initial draft

