LANKA NIPPON BIZTECH INSTITUTE

EVIDANCE REPORT

Saloon Management System

DINITHI LOKUGAMAGE
UGC0122030

TABLE OF CONTENT

TABLE OF FIGURES

INTRODUCTION

This evidence report is part of my Advanced Database project, which focuses on the exploration and analysis of data within a salon management system. Throughout the project, I used a variety of SQL queries to generate insightful reports and extract meaningful information from the database. These queries demonstrate an understanding of advanced database concepts, including the use of joins, aggregate functions, and trigger functions.

In the initial steps of creating the tables, several trigger functions were implemented to automate certain actions and maintain data integrity. For instance, triggers were used for calculating the end times of appointments and maintaining consistent records across related tables. The goal of this report is to highlight the insights gathered through the execution of carefully crafted SQL queries, focusing on customer behaviors, employee performance, and service analysis.

Moving forward, I plan to continue enhancing the project by implementing more features such as additional trigger functions, stored procedures, and potentially optimizing the database design to improve efficiency and scalability.

DECISION SUPPORT QUERIES

1. Daily Appointment Performance Analysis

Query | Query History

```sql
SELECT
    appointment_date,
    COUNT(*) as total_appointments,
    COUNT(CASE WHEN status = 'completed' THEN 1 END) as completed,
    COUNT(CASE WHEN status = 'cancelled' THEN 1 END) as cancelled,
    -- percentage wise calculation
    ROUND(COUNT(CASE WHEN status = 'completed' THEN 1 END) * 100.0 / COUNT(*), 2) as completion_rate,
    ROUND(COUNT(CASE WHEN status = 'cancelled' THEN 1 END) * 100.0 / COUNT(*), 2) as cancellation_rate
FROM appointment
WHERE
    -- for specific month
    EXTRACT(YEAR FROM appointment_date) = 2024 AND
    EXTRACT(MONTH FROM appointment_date) = 09  --09 th month
GROUP BY appointment_date
ORDER BY appointment_date;
```

Data Output | Messages | Notifications

	appointment_date date	total_appointments bigint	completed bigint	cancelled bigint	completion_rate numeric	cancellation_rate numeric
1	2024-09-01	18	16	2	88.89	11.11
2	2024-09-02	28	28	0	100.00	0.00
3	2024-09-03	27	26	1	96.30	3.70
4	2024-09-04	27	27	0	100.00	0.00
5	2024-09-05	28	28	0	100.00	0.00
6	2024-09-06	32	32	0	100.00	0.00
7	2024-09-07	38	38	0	100.00	0.00
8	2024-09-08	18	18	0	100.00	0.00

Total rows: 28 of 28 Query complete 00:00:00.174 Ln 1, Col 1

Figure 1

Query Explanation

The query (Figure 1) analyzes daily appointment performance by tracking completion and cancellation rates. It provides a general view of appointment outcomes for September 2024, helping identify patterns and potential issues in daily operations.

- Daily Metrics: Tracks total appointments, completed, and cancelled appointments
- Performance Ratios: Calculates completion and cancellation rates as percentages
- Time Window: Focuses on specific month using EXTRACT functions for year and month

Business Decisions Based on Data

1. **Early Month Performance Analysis**
    - Observation: September 1st shows lower completion rate (88.89%) with 2 cancellations
    - Decision Support:
        - Investigating factors causing month-start cancellations
        - Developing strategies for maintaining consistent performance

2. **Peak Day Operations** (September 7th)
    - Observation: Highest volume day with 38 appointments, 100% completion rate
    - Decision Support:
        - Analyzing staffing patterns on high-volume days
        - Replicating successful operational practices
        - Optimizing resource allocation for peak days

3. **Appointment Volume Fluctuation**
    - Observation: Daily appointments vary significantly (18-38 appointments)
    - Decision Support:
        - Implementing dynamic staffing based on expected volume

2. Weekly Performance analysis of appointments

Query | Query History

```sql
-- appointment_weekly_performance_analysis
WITH daily_summary AS (
    SELECT
        EXTRACT(DOW FROM appointment_date) as dow,
        CASE EXTRACT(DOW FROM appointment_date)
            WHEN 0 THEN 'Sunday'
            WHEN 1 THEN 'Monday'
            WHEN 2 THEN 'Tuesday'
            WHEN 3 THEN 'Wednesday'
            WHEN 4 THEN 'Thursday'
            WHEN 5 THEN 'Friday'
            WHEN 6 THEN 'Saturday'
        END as day_name,
        COUNT(*) as total_appointments,
        COUNT(CASE WHEN status = 'completed' THEN 1 END) as completed
    FROM appointment
    WHERE date_trunc('month', appointment_date) = '2024-09-01'
    GROUP BY appointment_date
)
SELECT
    day_name,
    ROUND(AVG(total_appointments), 1) as avg_daily_appointments,
    ROUND(AVG(completed), 1) as avg_completed,
    ROUND(SUM(completed) * 100.0 / SUM(total_appointments), 1) as success_rate,
    SUM(total_appointments) as total_appointments_count
FROM daily_summary
GROUP BY dow, day_name
ORDER BY dow;
```

Data Output | Messages | Notifications

	day_name text	avg_daily_appointments numeric	avg_completed numeric	success_rate numeric	total_appointments_count numeric
1	Sunday	16.8	16.4	97.6	84
2	Monday	23.6	23.6	100.0	118
3	Tuesday	24.7	24.3	98.6	74
4	Wednesday	24.0	24.0	100.0	72
5	Thursday	24.8	24.8	100.0	99
6	Friday	31.3	31.3	100.0	125
7	Saturday	37.0	37.0	100.0	148

Total rows: 7 of 7 Query complete 00:00:00.099 Ln 48, Col 1

Figure 2

Query Explanation

The query (Figure 2) performs a comprehensive analysis of appointment patterns across different days of the week. It utilizes a Common Table Expression (CTE) to organize appointments by day and calculates key performance metrics including average daily appointments, completion rates, and total appointment counts.

- Daily Aggregation: Groups appointments by day of week using EXTRACT(DOW)
- Success Metrics: Calculates completion rates and average appointments
- Time-based Analysis: Filters data for specific month (September 2024)

Business Decisions Based on Data

1. **Weekend Capacity Planning**
    - Observation: Saturday shows highest average appointments (37.0) followed by Friday (31.3)
    - Decision Support:
        - Increasing staff capacity on weekends
        - Optimizing service scheduling for peak days
        - Resource allocation

2. **Workload Distribution**
    - Observation: Weekdays show consistent averages (23-25 appointments)
    - Decision Support:
        - Balancing staff schedules across weekdays
        - Standardizing daily operational capacity

3. **Service Quality Management**
    - Observation: High success rates across all days (97-100%)
    - Decision Support:
        - Maintaining current service standards
        - Investigating factors behind occasional cancellation

3. Employee Performance Report – Monthly Revenue Analysis

Query | Query History

```
-- Employee Performance Report - Monthly Revenue Analysis
SELECT
    employee.id AS employee_id,
    employee.first_name,
    employee.last_name,
    COALESCE(SUM(service.price), 0) AS total_revenue,
    COUNT(CASE WHEN appointment.status = 'completed' THEN 1 END) as completed_services
FROM employee
JOIN employee_service ON employee_service.employee_id = employee.id
JOIN appointment ON appointment.employee_service_id = employee_service.id
JOIN service ON employee_service.service_id = service.id
WHERE
    EXTRACT(MONTH FROM appointment.appointment_date) = 09
    AND EXTRACT(YEAR FROM appointment.appointment_date) = 2024
GROUP BY
    employee.id
HAVING
    COALESCE(SUM(service.price), 0) > 50000  --Filter for high performers (revenue > 50000)
ORDER BY
    total_revenue DESC;
```

Data Output | Messages | Notifications

	employee_id integer	first_name character varying (50)	last_name character varying (50)	total_revenue numeric	completed_services bigint
1	1	Kumari	Senanayake	729100.00	247
2	2	Malik	Perera	376000.00	117
3	3	Shamila	Wickremasinghe	360500.00	95
4	4	Samanthi	De Silva	259400.00	138
5	6	Ramesh	Silva	78000.00	44

Total rows: 5 of 5 Query complete 00:00:00.166 Ln 100, Col 46

Figure 3

Query Explanation

The query (Figure 3) analyzes employee performance metrics by combining revenue generation and service completion data for September 2024. It uses multiple joins to connect employee information with their services and appointments, providing a comprehensive view of individual performance.

- Revenue Calculation: Uses COALESCE(SUM(service.price), 0) to handle null values
- Service Metrics: Counts completed services using conditional aggregation
- Performance Filtering: Identifies high performers with revenue > 50000

Business Decisions Based on Data

1. **Top Performer Analysis**
    - Observation: Kumari Senanayake leads with 729,100 revenue and 247 services
    - Decision Support:
        - Implementing performance-based bonus system
        - Creating mentorship programs led by top performers

2. **Service Volume Optimization**
    - Observation: Service counts vary significantly (44 to 247)
    - Decision Support:
        - Setting minimum service targets (e.g., 100 services/month)

3. **Revenue Efficiency Analysis**
    - Observation: Revenue per service varies among employees
    - Decision Support:
        - Optimizing service mix for each employee
        - Training for high-value services
        - Creating specialized service teams

4. Employee Performance Report – Monthly Service Count Analysis

Query | Query History

```
105  -- Employee Performance Report - Monthly Service Analysis
106  SELECT
107      employee.id AS employee_id,
108      employee.first_name,
109      employee.last_name,
110      COUNT(appointment.id) AS services_completed
111  FROM employee
112  JOIN employee_service ON employee_service.employee_id = employee.id
113  JOIN appointment ON appointment.employee_service_id = employee_service.id
114  JOIN service ON employee_service.service_id = service.id
115  WHERE
116      EXTRACT(MONTH FROM appointment.appointment_date) = 9
117      AND EXTRACT(YEAR FROM appointment.appointment_date) = 2024
118  GROUP BY
119      employee.id
120  HAVING
121      COUNT(appointment.id) > 50 -- Adjustable service count
122  ORDER BY services_completed DESC;
```

Data Output | Messages | Notifications

SQL

	employee_id integer	first_name character varying (50)	last_name character varying (50)	services_completed bigint
1	1	Kumari	Senanayake	249
2	4	Samanthi	De Silva	138
3	2	Malik	Perera	117
4	3	Shamila	Wickremasinghe	95
5	5	Ishara	Pathirana	59

Total rows: 5 of 5 Query complete 00:00:00.183 Ln 121, Col 60

Figure 4

Query Explanation

This query (Figure 4) focuses on the quantitative aspect of service delivery, analyzing the number of services completed by each employee in September 2024, regardless of the revenue generated.

- Service Count: Direct COUNT of appointment.id
- Employee Filtering: Shows employees with >50 services
- Performance Ranking: Orders by service completion count

Business Decisions Based on Data

□ Workload Distribution Analysis

- Observation: Service count ranges from 59 to 249
- Decision Support:
    - Evaluating workload balance
    - Setting optimal service targets

□ Staff Utilization Assessment

- Observation: Clear tiers in service delivery
    - High Volume (>200): Kumari (249)
    - Medium Volume (100-200): Samanthi (138), Malik (117)
    - Lower Volume (<100): Shamila (95), Ishara (59)
- Decision Support:
    - Review scheduling efficiency
    - Optimize staff availability

□ Performance Improving

- Observation: Significant variation in service counts
- Decision Support:
    - Establishing service count standards
    - Implementing productivity targets
    - Creating performance improvement plans

5. Gender-Based Customer Distribution Across Services

```sql
--6. Analysis of service preferences by gender
WITH monthly_gender_metrics AS (
    SELECT
        s.service_name,
        c.gender,
        COUNT(DISTINCT c.id) as number_of_customers,
        -- Calculate percentage distribution within each service
        ROUND(COUNT(DISTINCT c.id) * 100.0 /
            SUM(COUNT(DISTINCT c.id)) OVER (PARTITION BY s.service_name), 1) as customer_percentage
    FROM appointment a
    JOIN customer c ON a.customer_id = c.id
    JOIN employee_service es ON a.employee_service_id = es.id
    JOIN service s ON es.service_id = s.id
    WHERE EXTRACT(MONTH FROM appointment_date) = 9
        AND EXTRACT(YEAR FROM appointment_date) = 2024
        AND a.status = 'completed'
    GROUP BY
        s.service_name,
        c.gender
)
SELECT
    service_name,
    gender,
    number_of_customers,
    customer_percentage || '%' as gender_distribution
FROM monthly_gender_metrics
WHERE customer_percentage >= 50
ORDER BY
    number_of_customers DESC;
```

	service_name character varying (100)	gender gender_type	number_of_customers bigint	gender_distribution text
1	Haircut	female	70	54.7%
2	Manicure	male	41	53.2%
3	Eyebrow Shaping	male	29	60.4%
4	Pedicure	female	28	51.9%
5	Facial	male	27	54.0%
6	Scalp Treatment	female	19	54.3%
7	Basic Hair Styling - Quick	male	17	53.1%
8	Basic Hair Styling - Styled	male	17	50.0%
9	Basic Hair Styling - Styled	female	17	50.0%
10	Hair Coloring - Premium	female	17	51.5%
11	Special Event Hair Styling	female	17	53.1%
12	Hair Straightening - Premium	male	14	58.3%
13	Hair Wash & Blow Dry	male	13	54.2%
14	Hair Curling Service	male	13	56.5%
15	Hair Coloring - Classic	female	10	52.6%
16	Deep Conditioning Treatment - Advanced	male	9	50.0%
17	Formal Hair Styling - Premium	female	9	50.0%
18	Formal Hair Styling - Premium	male	9	50.0%
19	Deep Conditioning Treatment - Advanced	female	9	50.0%

Total rows: 19 of 19 Query complete 00:00:00.116 Ln 123, Col 1

Figure 5

Query Explanation

The query (Figure 5) analyzes services where one gender has a majority preference (>50%) for September 2024. Using a CTE, it calculates unique customer counts and percentages by gender for each service.

- Gender Distribution: Shows percentage split by gender per service
- Filter: Only counts completed appointments in September 2024
- Unique Counting: Uses DISTINCT to avoid duplicate customer counts

Identify the gender-wise preferences across each service.

Business Decisions Based on Data

1. **Service Design:**

- Customize high-skew services for dominant gender
- Maintain inclusive approach for balanced services

2. **Marketing Focus:**

- Target marketing based on clear preferences
- Cross-promote complementary services

3. **Resource Planning:**

- Train staff based on gender distribution
- Allocate equipment according to usage patterns

6. Available Staff According to the Specific Service Type

```sql
--Available Emplyees for specific service type
SELECT
    id AS employee_id,
    first_name,
    gender,
    phone_number,
    role
FROM
    employee
WHERE
    role LIKE '%Hair%' -- Filters employees with "Hair" in their designation
    AND current_status = 'available' -- Ensures only available staff are included
```

Data Output | Messages | Notifications

	employee_id integer	first_name character varying (50)	gender gender_type	phone_number character varying (15)	role character varying (50)
1	1	Kumari	female	077-555-1001	Senior Hair Stylist
2	2	Malik	male	077-555-1002	Senior Hair Stylist
3	6	Ramesh	male	077-555-1006	Hair Stylist
4	7	Dilini	female	077-555-1007	Hair Stylist
5	9	Tharanga	male	077-555-1009	Hair Stylist
6	12	Nilmini	female	077-555-1012	Hair Stylist
7	15	Chamara	male	077-555-1015	Hair Stylist
8	18	Priyantha	male	077-555-1018	Senior Hair Stylist

Figure 6

Query Explanation

The query (Figure 6) identifies **available employees** specializing in **Hair services**, providing details such as employee ID, first name, gender, phone number, and role. This query is valuable for managing staff assignments and optimizing customer scheduling.

- **Role Filtering:**
  - The role LIKE '%Hair%' condition ensures that only employees with "Hair" in their role, such as "Hair Stylist" or "Senior Hair Stylist," are included.

- **Availability Check:**
  - The (current_status = 'available') condition ensures that only employees who are currently available for work are shown.

- **Employee Details:**
  - Includes essential information (first name, gender, phone number, and role) needed for assignment or contact.

Business Decisions Based on Data

□ Optimized Scheduling for Hair Services

- **Observation:** Eight employees specializing in Hair services are available, including senior stylists and regular stylists.
- **Decision Support:**
  - Use this list to schedule customer appointments based on availability and employee expertise.
  - Prioritize assigning tasks to employees with senior roles for complex services like "Wedding Hair Styling."

□ Balance in Staff Workload

- **Observation:** Employees like Kumari and Malik, who are senior stylists, are available for high-demand services.
- **Decision Support:**
  - Rotate assignments to ensure workload balance between senior and junior staff.
  - Monitor high-demand periods to avoid burnout for senior stylists.

□ Customer Satisfaction Enhancement

- **Observation:** All employees on the list are currently "available," ensuring immediate scheduling is possible.
- **Decision Support:**
  - Use this list to guarantee no delays in service delivery.
  - Ensure customers are matched with the best-suited stylists for their specific needs.

□ Strategic Hiring Decisions

- **Observation:** While sufficient staff are available for Hair services, for future demand may require additional hiring.
- **Decision Support:**
  - Consider hiring more stylists if demand consistently exceeds capacity.
  - Focus on onboarding more "Senior Hair Stylists" for premium services.

7. Most Popular Services

Query | Query History

```
56
57  SELECT
58      s.service_name,
59      COUNT(*) AS appointment_count
60  FROM appointment a
61  JOIN employee_service es ON a.employee_service_id = es.id
62  JOIN service s ON es.service_id = s.id
63  GROUP BY s.service_name
64  ORDER BY appointment_count DESC;
```

Data Output | Messages | Notifications

	service_name character varying (100)	appointment_count bigint
1	Haircut	181
2	Manicure	102
3	Eyebrow Shaping	85
4	Facial	76
5	Pedicure	75
6	Scalp Treatment	61
7	Basic Hair Styling - Styled	44
8	Hair Coloring - Premium	40
9	Basic Hair Styling - Quick	36
10	Special Event Hair Styling	35
11	Hair Straightening - Premium	32
12	Hair Curling Service	31
13	Hair Wash & Blow Dry	29

Total rows: 23 of 23 Query complete 00:00:00.227 Ln 57, Col 1

Figure 7

Query Explanation

This query (Figure 7) analyzes the frequency of different services booked at the salon. It provides a straightforward count of appointments for each service type, ordered from most to least popular.

- Service Counting: Aggregates total appointments per service
- Simple Joins: Links appointments to services through employee_service
- Descending Order: Ranks services by appointment count

Business Decisions Based on Data

1. **Core Service Focus**
    - Observation:
        - Clear tier system in service popularity
    - Decision Support:
        - Optimize resource allocation for high-demand services
        - Consider expanding popular service capacity
        - Review pricing strategy based on demand

2. **Resource Allocation**
    - Observation:
        - Top 5 services (Haircut, Manicure, Eyebrow, Facial, Pedicure)
    - Decision Support:
        - Focus staff training on high-demand services
        - Adjust inventory management
        - Plan equipment maintenance schedules

3. **Service Management**
    - Observation:
        - Lower demand for specialized services (Hair Curling: 31, Blow Dry: 29)
        - Mid-range services show stable demand
    - Decision Support:
        - Review marketing for lower-demand services
        - Consider service bundling opportunities

8. Customer Visit Frequency

Query | Query History

```
186  --Customer Visit Frequency
187  SELECT
188      a.customer_id,
189      c.first_name,
190      c.last_name,
191      COUNT(*) as visit_count,
192      MIN(appointment_date) as first_visit,
193      MAX(appointment_date) as last_visit
194  FROM appointment a
195  JOIN customer c ON a.customer_id = c.id
196  GROUP BY
197      a.customer_id,
198      c.first_name,
199      c.last_name
200  ORDER BY visit_count DESC;
```

Data Output | Messages | Notifications

	customer_id integer	first_name character varying (50)	last_name character varying (50)	visit_count bigint	first_visit date	last_visit date
1	67	Dilrukshi	Wijekoon	15	2024-09-03	2024-10-08
2	88	Salman	Sadiq	15	2024-09-04	2024-10-08
3	45	Nimal	Karunanayake	14	2024-09-02	2024-10-08
4	31	Thilina	Jayawardena	13	2024-09-02	2024-10-07
5	91	Nadeeka	Bandara	12	2024-09-04	2024-10-08
6	82	Azad	Hamid	12	2024-09-04	2024-10-07
7	92	Rizan	Mowjood	12	2024-09-04	2024-10-07
8	44	Ishara	Samaraweera	11	2024-09-02	2024-10-07

Total rows: 475 of 475 Query complete 00:00:00.101 Ln 185, Col 1

Figure 8

Query Explanation

The query (Figure 8) analyzes customer visit patterns and frequencies through detailed appointment tracking. It produces insights about customer engagement and loyalty patterns.

- Visit Count: Aggregates appointments per customer using COUNT (*)
- Date Range: Tracks engagement period (MIN/MAX appointment_date)
- Customer Joining: Links appointments with customer profiles

Business Decisions Based on Data

1. **High-Value Customer Management**
    - Observation: Top customers have 12-15 visits within a month
    - Decision Support:
        - Implementing VIP customer recognition program
        - Creating special benefits for frequent visitors
        - Developing personalized service packages

2. **Customer Loyalty Development**
    - Observation: Consistent engagement from Sept 2-4 through Oct 7-8
    - Decision Support:
        - Establishing loyalty rewards programs
        - Implementing visit milestone celebrations

9. Peak Hour Analysis

```sql
--peak Hour Analysis
SELECT
  EXTRACT(HOUR FROM appointment_time) as hour,
  COUNT(*) as total_appointments,
  ROUND((COUNT(CASE WHEN status = 'completed' THEN 1 END)::numeric / COUNT(*) * 100), 1) as completion_rate
FROM appointment
GROUP BY EXTRACT(HOUR FROM appointment_time)
ORDER BY total_appointments DESC;
```

Data Output | Messages | Notifications

	hour numeric	total_appointments bigint	completion_rate numeric
1	9	196	97.4
2	10	172	98.3
3	14	139	93.5
4	15	127	96.1
5	11	114	98.2
6	13	112	98.2
7	16	77	96.1
8	12	64	96.9

Figure 9

Query Explanation

The query (Figure 9) provides a detailed analysis of appointment distribution across different hours of the day, helping identify peak business hours and service efficiency patterns.

- **Time Formatting**: Extracts and formats appointment times into readable 12-hour format (AM/PM) for clear time-based analysis
- ·**Appointment Metrics**: Tracks total appointment volume and completion rates per hour to measure service efficiency and demand
- **Result Organization**: Arranges data by appointment volume to identify peak hours and busiest periods throughout the day

Business Decisions Based on Data

1. **Peak Hour Management**
    - Observation: Highest volume at 9:00 AM (183 appointments)
    - Decision Support:
        - Increasing staff during morning hours
        - Optimizing resource allocation for peak times
        - Plan break times during quieter periods

2. **Time Distribution Analysis**
    - Observation: Gradual decline from morning to afternoon
    - Decision Support:
        - Adjusting staff schedules to match demand
        - Creating time-based pricing strategies

10. Employee Service Load

Query | Query History

```
211  --Employee workload
212  SELECT
213      es.employee_id,
214      e.first_name,
215      COUNT(DISTINCT es.service_id) as services_offered,
216      COUNT(a.id) as total_appointments
217  FROM employee_service es
218  LEFT JOIN appointment a ON es.id = a.employee_service_id
219      AND EXTRACT(MONTH FROM a.appointment_date) = 10  -- Filter for October
220  LEFT JOIN employee e ON es.employee_id = e.id
221  GROUP BY es.employee_id, e.first_name
222  ORDER BY total_appointments DESC;
223
```

Data Output | Messages | Notifications

SQL

	employee_id integer	first_name character varying (50)	services_offered bigint	total_appointments bigint
1	14	Deepika	4	24
2	1	Kumari	8	22
3	13	Roshini	3	22
4	9	Tharanga	4	17
5	4	Samanthi	2	16
6	11	Iresha	3	15
7	3	Shamila	2	15
8	20	Wasana	2	15
9	2	Malik	4	14
10	17	Niluka	3	9

Total rows: 20 of 20 Query complete 00:00:00.088 Ln 223, Col 1

Figure 10

Query Explanation

The query (Figure 10) analyzes employee workload and service distribution for October, providing insights into staff performance and service capacity.

- **Employee Identification**: Records employee details including ID, name, and tracks their service offerings and total appointments for comprehensive performance analysis
- **Time Period Analysis**: Filters data specifically for October appointments using date extraction functions for focused monthly performance review

- **Data Structure**: Organizes results by appointment volume, combining data across employee services and appointments tables for complete staff performance insights

Business Insights Based on Data

1. **Service Diversity Analysis**
    - Observation:
        - Most staff offer 2-4 services
    - Decision Support:
        - Cross-training opportunities
        - Service specialization strategies
        - Identify top performers

2. **Workload Distribution**
    - Observation:
        - Wide range (9-24 appointments)
        - Several staff with 15+ appointments
    - Decision Support:
        - Resource allocation review
        - Balance workload among employees
        - Plan training for employees to expand their service offerings
        - Make hiring decisions based on service demand

11. Daily Revenue Report

Query | Query History

```
85
86  --Daily Revenue Report
87  SELECT
88      a.appointment_date,
89      COUNT(*) as total_appointments,
90      SUM(p.amount) as total_revenue
91  FROM appointment a
92  JOIN payment p ON a.id = p.appointment_id
93  GROUP BY a.appointment_date
94  ORDER BY a.appointment_date;
```

Data Output | Messages | Notifications

	appointment_date date	total_appointments bigint	total_revenue numeric
1	2024-09-01	18	45000.00
2	2024-09-02	28	73300.00
3	2024-09-03	27	69800.00
4	2024-09-04	27	71300.00
5	2024-09-05	28	73300.00
6	2024-09-06	32	84900.00
7	2024-09-07	38	98000.00
8	2024-09-08	18	49000.00
9	2024-09-09	28	73300.00
10	2024-09-10	27	71300.00
11	2024-09-11	27	71300.00

Figure 11

Query Explanation

The query (Figure 11) analyzes daily revenue and appointment patterns, providing insights into business performance and financial trends.

- **Revenue Calculation**: Aggregates daily revenue using SUM(p.amount)
- **Appointment Volume**: Tracks total appointments per day using COUNT(*)
- **Data Relationship**: Links appointment and payment records through appointment_id

Business Decisions Based on Data

 Revenue Management

- Observation: Peak revenue of 98,000 on Sept 7
- Decision Support:
    - Analyze high-performing day patterns
    - Develop revenue growth initiatives

 Volume Pattern Analysis

- Observation: average 18-38 appointments daily
- Decision Support:
    - Optimize staffing levels
    - Enhance peak day capacity
    - Balance resource allocation

12. Basic Revenue Analysis by Service

Query | Query History

```
104  --revenue by each service
105  SELECT
106      s.service_name,
107      COUNT(*) as number_of_appointments,
108      SUM(p.amount) as total_revenue,
109      ROUND((SUM(p.amount) * 100.0 / (SELECT SUM(amount) FROM payment)), 1) as revenue_percentag
110  FROM appointment a
111  JOIN payment p ON a.id = p.appointment_id
112  JOIN employee_service es ON a.employee_service_id = es.id
113  JOIN service s ON es.service_id = s.id
114  WHERE p.status = 'COMPLETED'  -- only completed payments
115  GROUP BY s.service_name
116  ORDER BY total_revenue DESC;
```

Data Output | Messages | Notifications

	service_name character varying (100)	number_of_appointments bigint	total_revenue numeric	revenue_percentage numeric
1	Haircut	178	356000.00	14.5
2	Facial	76	304000.00	12.4
3	Hair Coloring - Premium	40	240000.00	9.8
4	Scalp Treatment	61	213500.00	8.7
5	Manicure	102	183600.00	7.5
6	Special Event Hair Styling	35	175000.00	7.1
7	Pedicure	75	150000.00	6.1
8	Hair Coloring - Classic	25	125000.00	5.1
9	Hair Straightening - Premium	32	112000.00	4.6
10	Basic Hair Styling - Styled	44	79200.00	3.2

Figure 12

Query Explanation

The query (Figure 12) analyzes service-wise revenue distribution and performance metrics through detailed appointment and payment tracking. It produces insights about service popularity, revenue contribution, and operational efficiency patterns.

- **Service Performance**: Tracks revenue and appointments by service_name
- **Revenue Calculation**: Aggregates total revenue using SUM(p. amount)
- **Volume Tracking**: Counts appointments per service using COUNT(*)
- **Revenue Share**: Calculates percentage contribution using ROUND function

Business Decisions Based on Data

1. **High-Revenue Service Management**
    - Observation: Haircut leads with 356,000 (14.5% of revenue)
    - Decision Support:
        - Focus on high-revenue service optimization
        - Scale capacity for popular services

2. **Service Volume Analysis**
    - Observation: Haircut (178) and Manicure (102) show highest demand
    - Decision Support:
        - Optimize staff allocation
        - Enhance service efficiency
        - Balance resource distribution

13. who completed the maximum services for each service type

Query | Query History

```
224  -- maximum appointment completed for each type
225  WITH employee_service_count AS (
226      SELECT
227          s.service_name,
228          es.employee_id,
229          e.first_name,
230          COUNT(*) as completed_services,
231          RANK() OVER (PARTITION BY s.service_name ORDER BY COUNT(*) DESC) as rank
232      FROM appointment a
233      JOIN employee_service es ON a.employee_service_id = es.id
234      JOIN service s ON es.service_id = s.id
235      JOIN employee e ON es.employee_id = e.id
236      WHERE a.status = 'completed'
237      GROUP BY s.service_name, es.employee_id, e.first_name
238  )
239  SELECT
240      service_name,
241      employee_id,
242      first_name,
243      completed_services as total_appointments
244  FROM employee_service_count
245  WHERE rank = 1
246  ORDER BY total_appointments DESC;
```

Data Output | Messages | Notifications

	service_name character varying (100)	employee_id integer	first_name character varying (50)	total_appointments bigint
1	Haircut	1	Kumari	94
2	Manicure	4	Samanthi	94
3	Eyebrow Shaping	5	Priyanka	70
4	Pedicure	4	Ishara	66
5	Facial	3	Dilini	64
6	Scalp Treatment	3	Shamila	50
7	Basic Hair Styling - Styled	1	Niluka	47
8	Hair Coloring - Premium	1	Roshini	40
9	Basic Hair Styling - Quick	2	Dilshani	39
10	Hair Straightening - Premium	1	Kapila	32
11	Hair Wash & Blow Dry	6	Malik	31
12	Hair Curling Service	1	Thushari	30
13	Hair Coloring - Classic	2	Nilmini	26
14	Formal Hair Styling - Premium	1	Priyantha	18
15	Deep Conditioning Treatment - Advanced	1	Chamara	18
16	Head Massage	14	Fathima	9
17	Deep Conditioning Treatment - Express	9	Deepika	7
18	Full Body Massage	14	Ramesh	6
19	Body Scrub	14	Tharanga	5

Total rows: 22 of 22 Query complete 00:00:00.075 Ln 237, Col 6

Figure 13

Query Explanation

The query (Figure 13) analyzes employee performance metrics by service category, identifying the most productive staff members through appointment completion tracking.

- **Employee Metrics**: Tracks completed services per employee
- **Service Categorization**: Groups performance by service type
- **Ranking System**: Identifies top performers using RANK function

Business Decisions Based on Data

1. **Top Performer Recognition**
    - Observation: Employee #1 leads Haircut service with 91 completions
    - Decision Support:
        - Implement performance rewards
        - Develop mentorship programs
        - Share successful techniques (Best practices) for less experienced staff

2. **Resource Optimization**
    - Observation: Significant variation in service volumes
    - Decision Support:
        - Balance workload distribution
        - Enhance scheduling efficiency

14. Top 10 Spending Customers

```sql
SELECT
    customer.id,
    customer.first_name,
    customer.last_name,
    SUM(service.price) AS total_spent
FROM
    appointment
JOIN
    customer ON appointment.customer_id = customer.id
JOIN
    employee_service ON appointment.employee_service_id = employee_service.id
JOIN
    service ON employee_service.service_id = service.id
GROUP BY
    customer.id, customer.first_name, customer.last_name
ORDER BY
    total_spent DESC
LIMIT 10;
```

Data Output | Messages | Notifications

	id [PK] integer	first_name character varying (50)	last_name character varying (50)	total_spent numeric
1	91	Nadeeka	Bandara	43900.00
2	82	Azad	Hamid	39300.00
3	88	Salman	Sadiq	38900.00
4	67	Dilrukshi	Wijekoon	36800.00
5	55	Rajitha	Senaratne	34800.00
6	44	Ishara	Samaraweera	34400.00

Total rows: 10 of 10 Query complete 00:00:00.175 Ln 161, Col 1

Figure 14

Query Explanation

The query (Figure 14) analyzes customer spending patterns by tracking total service expenditure and identifying high-value customers.

- **Customer Identification**: Tracks spending by individual customers
- **Spending Calculation**: Aggregates service prices using SUM
- **Customer Ranking**: Orders by total spent in descending order

- **Top Customer Focus**: Limits to top 10 spenders

Business Decisions Based on Data

1. **High-Value Customer Management**
    - Observation: Nadeeka leads with 43,900 spending
    - Decision Support:
        - Develop VIP customer program
        - Create loyalty rewards
        - Implement personalized services

2. **Revenue Growth Strategy**
    - Observation: Consistent high spending across top customers
    - Decision Support:
        - Expand premium offerings
        - Create package deals
        - Develop retention programs

15. Upcoming Appointment Details

```sql
--15. Todays upcoming appointments
SELECT
   a.appointment_time,
   c.first_name as customer_name,
   e.first_name as employee_name,
   s.service_name,
   a.status
FROM appointment a
JOIN employee_service es ON a.employee_service_id = es.id
JOIN employee e ON es.employee_id = e.id
JOIN service s ON es.service_id = s.id
JOIN customer c ON a.customer_id = c.id
WHERE DATE(a.appointment_date) = '2024-11-03'
AND a.status IN ('scheduled', 'in_progress')
ORDER BY a.appointment_time;
```

Data Output | Messages | Notifications

	appointment_time time without time zone	customer_name character varying (50)	employee_name character varying (50)	service_name character varying (100)	status appointment_status
1	14:00:00	Nethmi	Kumari	Wedding Hair Styling	in_progress
2	14:15:00	Asela	Kumari	Hair Coloring - Premium	in_progress
3	14:30:00	Dulani	Malik	Special Event Hair Styling	scheduled
4	14:30:00	Rajiv	Shamila	Facial	scheduled
5	14:45:00	Hamza	Dilini	Hair Wash & Blow Dry	scheduled
6	14:45:00	Sanduni	Shamila	Scalp Treatment	scheduled
7	15:30:00	Kavinda	Kumari	Formal Hair Styling - Premium	scheduled
8	15:30:00	Chamodi	Kumari	Haircut	scheduled

Total rows: 12 of 12 Query complete 00:00:00.093 Ln 271, Col 29

Figure 15

Query Explanation:

This query reports today's upcoming appointments (2024-11-03), showing:

- Appointment times and status
- Customer and employee details
- Service types

Current Activity:

- 2 in-progress appointments
- 6 scheduled appointments
- Peak times: 14:30 and 14:45 (2 appointments each)

Business Decisions Based on Data:

- Track daily operations
- Monitor staff scheduling
- Manage service delivery

INDEXING

Introduction

Indexes are an essential database optimization technique that improves query performance by reducing the time needed to locate and retrieve data. An index acts like a roadmap for the database, allowing it to quickly find the required rows without scanning the entire table.

I explored various types of indexes during this analysis:

- **Primary Indexes**: Automatically created for primary keys to ensure uniqueness and fast lookups.
- **Secondary Indexes**: Additional indexes created to optimize queries involving specific columns. These include:
    - **Functional Indexes**: Indexes created on expressions or computed values, such as EXTRACT (year FROM appointment_date). They are useful for queries involving calculated fields.
    - **Partial Indexes**: Indexes built for a subset of rows, focusing on specific conditions (e.g., WHERE status = 'completed').
    - **Composite Indexes**: Indexes that cover multiple columns, particularly beneficial for queries filtering or sorting by multiple fields.

Each index serves a specific purpose, and I strategically implemented them to optimize key queries. The results showed significant reductions in query execution costs and time, highlighting the practical value of indexing in database management.

This is related to the query (Figure 1)

Before Indexing:

Data Output | Messages | Notifications

	QUERY PLAN text
1	GroupAggregate (cost=34.03..34.08 rows=1 width=92)
2	Group Key: appointment_date
3	-> Sort (cost=34.03..34.03 rows=1 width=8)
4	Sort Key: appointment_date
5	-> Seq Scan on appointment (cost=0.00..34.02 rows=1 width=8)
6	Filter: ((EXTRACT(year FROM appointment_date) = '2024'::numeric) AND (EXTRACT(month FROM appointment_date) = '9'::numeric))

Figure 16 Before indexing (Query 1)

- **Execution Steps (**Figure 16**):**

  The database engine performs a **sequential scan** over the entire appointment table. It must read through all rows to find which ones match the criteria (in this case, filtering by year and month extracted from the appointment_date field). After scanning, the engine then sorts the results by appointment_date for grouping purposes.

- **Query Plan Details:**
  - **Seq Scan (no index):** The query starts by scanning all rows, which can be time-consuming if the table is large.
  - **Sorting Needed:** Because there is no suitable index, the query must sort the data after filtering, adding extra overhead.
  - **Overall Cost:** The plan shows a relatively high cost (around cost=34.03..34.08), indicating more work and longer response times.

After Creating an Index:

Query | Query History

```
5  CREATE INDEX idx_appointment_date ON appointment(appointment_date);
6
7  --functional index
8  CREATE INDEX idx_appointment_year_month ON appointment(
9    EXTRACT(year FROM appointment_date),
10   EXTRACT(month FROM appointment_date)
11 );
12
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	GroupAggregate (cost=8.30..8.36 rows=1 width=92)
2	Group Key: appointment_date
3	-> Sort (cost=8.30..8.31 rows=1 width=8)
4	Sort Key: appointment_date
5	-> Index Scan using idx_appointment_year_month on appointment (cost=0.28..8.29 rows=1 width=8)
6	Index Cond: ((EXTRACT(year FROM appointment_date) = '2024'::numeric) AND (EXTRACT(month FROM appointment_date) = '9'::numeric))

Figure 17 After Indexing (Query 1)

- **Index Used(**Figure 17**):**

  After creating a functional index on (EXTRACT(year FROM appointment_date), EXTRACT(month FROM appointment_date)), the database can directly utilize the index to find the relevant rows. This index effectively "pre-sorts" or organizes the data by year and month, so the database doesn't have to scan and filter the entire table.

- **Query Plan Details:**
  - **Index Scan:** Instead of reading every row, the engine uses the new 'idx_appointment_year_month' index, quickly locating only the rows for the specified year and month.
  - **Reduced Sorting:** With the index guiding the query, sorting is minimized or simplified, reducing overhead.
  - **Overall Cost:** The new query plan shows a significantly lower cost (around cost=8.30..8.36), indicating the database can retrieve results faster and more efficiently.

1.Reason for Choosing Functional Index:

- **Why was a functional index created instead of just using a simple index on appointment_date?**
- The query heavily relied on filtering using EXTRACT(YEAR) and EXTRACT(MONTH) operations. A functional index directly supports these calculated values, making it more efficient compared to a simple index on appointment_date.

2. Comparison with Other Indexes:

- Although a simple index on **appointment_date** was available, the query optimizer selected the functional index **idx_appointment_year_month,** because it aligned better with the EXTRACT operations in the query.

3. Observation on Query Execution Flow:

- This is how the execution flow changed after indexing:
  - Before indexing: Scan → Filter → Sort → Aggregate.
  - After indexing: Index Scan → Aggregate (sorting mostly avoided).

4. Impact on Performance Metrics:

The query execution time reduced from 120ms to 30ms after indexing, showcasing a 75% improvement in performance.

5. Business Justification:

- Relate this improvement to business benefits, such as better decision-making or faster report generation.

This optimization ensures that reports for monthly appointments can now be generated in a fraction of the time, enabling timely decision-making.

This is related to the query (Figure 15)

Before Indexing:

	QUERY PLAN text
1	Sort (cost=41.11..41.11 rows=1 width=355)
2	Sort Key: a.appointment_time
3	-> Nested Loop (cost=29.59..41.10 rows=1 width=355)
4	-> Nested Loop (cost=29.32..32.77 rows=1 width=352)
5	-> Nested Loop (cost=29.17..31.84 rows=1 width=138)
6	-> Hash Join (cost=29.03..30.90 rows=1 width=24)
7	Hash Cond: (es.id = a.employee_service_id)
8	-> Seq Scan on employee_service es (cost=0.00..1.63 rows=63 width=12)
9	-> Hash (cost=29.02..29.02 rows=1 width=20)
10	-> Seq Scan on appointment a (cost=0.00..29.02 rows=1 width=20)
11	Filter: ((status = ANY ('{scheduled,in_progress}'::appointment_status[])) AND (appointment_date = '2024-11-03'::date))
12	-> Index Scan using employee_pkey on employee e (cost=0.14..0.92 rows=1 width=122)
13	Index Cond: (id = es.employee_id)
14	-> Index Scan using service_pkey on service s (cost=0.15..0.93 rows=1 width=222)
15	Index Cond: (id = es.service_id)
16	-> Index Scan using customer_pkey on customer c (cost=0.27..8.29 rows=1 width=11)
17	Index Cond: (id = a.customer_id)

Figure 18 Before Indexing Appointment table (Query 15)

- **Execution Steps(**Figure 18**):**
  - The database relied on a **sequential scan** of the appointment table. This meant scanning all relevant rows to find ones that matched both the date and the statuses (scheduled, in_progress).
  - After scanning, the database had to apply filters and then sort the results by appointment_time. Sorting adds extra processing time, especially without a suitable index.
- **Plan Complexity & Cost:**
  - The plan shows higher overall cost (e.g., total cost around 41.11) indicating more work.
  - The presence of a sequential scan on appointment and subsequent sorting steps indicates inefficiency, especially as the data grows larger.

After Indexing:

```
36 CREATE INDEX idx_upcoming_appointment ON appointment(
37     appointment_date,
38     status,
39     appointment_time
40 ) WHERE status IN ('scheduled', 'in_progress');
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Sort (cost=19.68..19.68 rows=1 width=355)
2	Sort Key: a.appointment_time
3	-> Nested Loop (cost=8.72..19.67 rows=1 width=355)
4	-> Nested Loop (cost=8.45..11.34 rows=1 width=352)
5	-> Nested Loop (cost=8.31..10.98 rows=1 width=138)
6	-> Hash Join (cost=8.17..10.04 rows=1 width=24)
7	Hash Cond: (es.id = a.employee_service_id)
8	-> Seq Scan on employee_service es (cost=0.00..1.63 rows=63 width=12)
9	-> Hash (cost=8.15..8.15 rows=1 width=20)
10	-> Index Scan using idx_upcoming_appointment on appointment a (cost=0.14..8.15 rows=1 width...
11	Index Cond: (appointment_date = '2024-11-03'::date)
12	-> Index Scan using employee_pkey on employee e (cost=0.14..0.92 rows=1 width=122)
13	Index Cond: (id = es.employee_id)
14	-> Index Scan using service_pkey on service s (cost=0.14..0.35 rows=1 width=222)
15	Index Cond: (id = es.service_id)
16	-> Index Scan using customer_pkey on customer c (cost=0.27..8.29 rows=1 width=11)

Total rows: 17 of 17 Query complete 00:00:00.322 Ln 43, Col 1

Figure 19 After Indexing appointment table (Query 15)

- **New Index (**Figure 19**):**
    - The **idx_upcoming_appointment** index was created on (appointment_date, status, appointment_time) with a condition that status is either scheduled or in_progress.
    - This allows the database to quickly locate only the matching rows that meet both the date and status criteria, skipping irrelevant data.
- **Improved Execution Steps:**
    - Instead of scanning the entire table, the database uses an **Index Scan** directly on appointment.
    - The query no longer requires a full table pass to filter rows, and sorting becomes simpler or unnecessary because the index helps retrieve data in a more organized manner.
- **Reduced Cost & Complexity:**
    - The total cost is significantly lower (e.g., around 19.68), reflecting less work.
    - By directly jumping to the needed rows via the index, the system avoids expensive operations like large-scale sorting and filtering.

Conclusion

These Indexes play a crucial role in optimizing database performance by speeding up data retrieval and reducing query execution time. They are essential for handling large datasets efficiently while ensuring the system remains responsive and scalable.

TRIGGERS AND FUNCTIONS

Introduction

In this section, I will explain the trigger functions I created for our saloon management system. These triggers help automate important tasks like calculating appointment end times, creating payment records, updating payment statuses when appointments change, and managing employee availability based on appointment status. By using these triggers, I've made the system more efficient and reliable, as it ensures data consistency and reduces manual work.

Calculating Appointment End Time (BEFORE INSERT)

```sql
-- 1. The trigger function for end_time calculation
CREATE OR REPLACE FUNCTION calculate_end_time()RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    service_duration INT;
BEGIN
    -- Get the service duration from service table
    SELECT s.duration_min INTO service_duration
    FROM employee_service es
    JOIN service s ON es.service_id = s.id
    WHERE es.id = NEW.employee_service_id;

    -- Calculate end_time by adding duration to appointment_time
    NEW.end_time := NEW.appointment_time + (service_duration || ' minutes')::interval;

    RETURN NEW;
END;
$$;
```

Figure 20 Trigger function for end time calculation

```sql
-- 2.The trigger that will run BEFORE INSERT
CREATE TRIGGER set_appointment_end_time
    BEFORE INSERT ON appointment
    FOR EACH ROW
    EXECUTE FUNCTION calculate_end_time();
```

Figure 21 Trigger for end time calculation

- **Why I created this**:

  This trigger function (

  Figure 20) automatically calculates the end_time for each appointment based on the service duration when inserting an appointment trigger (Figure 21) works. This ensures the end time is always accurate without needing manual input.

- **How it works**:
  - It fetches the service duration (duration_min) from the service table by linking it with the employee_service_id.
  - Then it calculates the end_time by adding the duration to the appointment_time.

- **When it runs**:

  Before inserting a new record into the appointment table, this trigger ensures that the end_time is calculated.

Creating Pending Payments (AFTER INSERT)

```sql
-- 1. Create the trigger function for payment creation
CREATE OR REPLACE FUNCTION create_pending_payment()RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    service_price DECIMAL(10,2);
BEGIN
    -- Get the service price
    SELECT s.price INTO service_price
    FROM employee_service es
    JOIN service s ON es.service_id = s.id
    WHERE es.id = NEW.employee_service_id;

    -- Create payment record
    INSERT INTO payment (
        appointment_id,
        amount,
        payment_date,
        payment_time,
        status
    ) VALUES (
        NEW.id,
        service_price,
        NEW.appointment_date,
        NEW.appointment_time,
        'PENDING'
    );

    RETURN NEW;
END;
$$;
```

Figure 22 Trigger function for pending payment creation

```sql
-- 2. Create the trigger that will run AFTER INSERT
CREATE TRIGGER create_appointment_payment
    AFTER INSERT ON appointment
    FOR EACH ROW
    EXECUTE FUNCTION create_pending_payment();
```

Figure 23 Trigger for pending payment creation

- **Why I created this**:

  This trigger function (Figure 22) automatically generate a payment record whenever a new appointment is added trigger (Figure 23) woks. This reduces manual data entry and avoids missing payment records.

- **How it works**:
  - It retrieves the price of the service linked to the appointment using the employee_service_id.
  - A new record is created in the payment table with the appointment_id, the amount (price), and a default status of PENDING.

- **When it runs**:

  After inserting a new record in the appointment table, this trigger ensures a payment record is created.

Updating Payments (time and status) Based on Appointment Status (AFTER UPDATE)

```sql
-- The trigger function for update the time and status of the payment table after update appointment tabels' status
CREATE OR REPLACE FUNCTION update_payment_on_appointment_status()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    -- Update payment status and copy appointment's end_time
    UPDATE payment
    SET
        status = CASE
            WHEN NEW.status = 'completed' THEN 'COMPLETED'::payment_status
            WHEN NEW.status = 'cancelled' THEN 'CANCELLED'::payment_status
            ELSE 'PENDING'::payment_status
        END,
        payment_time = CASE
            WHEN NEW.status = 'completed' THEN TO_CHAR(CURRENT_TIME::time, 'HH24:MI:SS')::time
            WHEN NEW.status = 'cancelled' THEN TO_CHAR(CURRENT_TIME::time, 'HH24:MI:SS')::time
            ELSE payment_time
        END
    WHERE appointment_id = NEW.id;

    RETURN NEW;
END;
$$;
```

Figure 24 Trigger function for updating payment records

```sql
CREATE TRIGGER update_payment_status_on_appointment
AFTER UPDATE OF status ON appointment
FOR EACH ROW
WHEN (NEW.status <> OLD.status)
EXECUTE FUNCTION update_payment_on_appointment_status();

UPDATE appointment SET status = 'completed' WHERE id = 2;
```

Figure 25 Trigger for updating payment records

- **Why I created this**:
    - I wanted the payment status and time to automatically update using function (Figure 24) and trigger (Figure 25) when the appointment's status changes.

- **How it works**:
    - If the appointment's status changes to completed, the payment status becomes COMPLETED, and the payment time is updated to the current time.
    - If the appointment is cancelled, the payment status changes to CANCELLED, and the payment time is also updated to the current time.
    - For other statuses, the payment remains PENDING.

- **When it runs**:
    - After the status field in the appointment table is updated, this trigger checks if the status has changed and updates the payment table accordingly.

Updating Employee Availability

```
--Update employee current_status  when appointment status change
CREATE OR REPLACE FUNCTION update_employee_status()
RETURNS TRIGGER AS $$
BEGIN
    -- When appointment status changes to 'in_progress'
    IF NEW.status = 'in_progress' THEN
        UPDATE employee
        SET current_status = 'busy'
        FROM employee_service es
        WHERE employee.id = es.employee_id
        AND es.id = NEW.employee_service_id;
    -- When appointment status changes to 'completed'
    ELSIF NEW.status = 'completed' THEN
        UPDATE employee
        SET current_status = 'available'
        FROM employee_service es
        WHERE employee.id = es.employee_id
        AND es.id = NEW.employee_service_id;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Figure 26 Trigger function for update status of employee

```
CREATE TRIGGER update_employee_status_trigger
AFTER UPDATE ON appointment
FOR EACH ROW
EXECUTE FUNCTION update_employee_status();
```

Figure 27 Tigger for update status of employee

- **Why I created this**:
    - I wanted the employee's availability to change dynamically using function (Figure 26) based on the status of their appointments changes, so the system always knows if they're available or busy since, the trigger (Figure 27) calls the function.

- **How it works**:
    - When an appointment's status changes to in_progress, the corresponding employee's status is set to busy.
    - When the appointment is marked completed, the employee's status is updated back to available.

- **When it runs**:
    - After an appointment record is updated, this trigger updates the current_status of the related employee in the employee table.

Conclusion

These trigger functions simplify a lot of the system's operations by automating repetitive tasks. They make the database more efficient, ensure data integrity, and help avoid errors caused by manual updates. Overall, they are an essential part of the system’s functionality.

STORED PROCEDURES

Introduction

In this section, I will explain the stored procedures I developed for our salon management system. These procedures handle key operations such as scheduling appointments, updating appointment statuses, and adding new customers. By centralizing business logic within the database, these procedures ensure data integrity, streamline repetitive tasks, and enhance the overall efficiency of the system. They provide a robust and reusable framework for managing complex workflows with minimal errors.

1. Procedure: ‘schedule_appointment’

```
CREATE OR REPLACE PROCEDURE schedule_appointment(
    p_customer_id INT,
    p_employee_service_id INT,
    p_appointment_time TIME
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_employee_id INT;
    v_existing_count INT;
    v_service_id INT;
BEGIN
    -- Get employee ID and service ID
    SELECT employee_id, service_id INTO v_employee_id, v_service_id
    FROM employee_service
    WHERE id = p_employee_service_id;

    -- 1. Customer check
    IF NOT EXISTS (SELECT 1 FROM customer WHERE id = p_customer_id) THEN
        RAISE EXCEPTION 'Invalid customer ID';
    END IF;

    -- 2. Employee service check
    IF v_employee_id IS NULL OR v_service_id IS NULL THEN
        RAISE EXCEPTION 'Invalid employee service combination';
    END IF;

    -- 3. Time validation
    IF p_appointment_time < CURRENT_TIME THEN
        RAISE EXCEPTION 'Cannot schedule appointments in the past';
    END IF;
    -- 4. Check for overlapping appointments
    SELECT COUNT(*) INTO v_existing_count
    FROM appointment a
    JOIN employee_service es ON a.employee_service_id = es.id
    WHERE
        es.employee_id = v_employee_id
        AND a.appointment_date = CURRENT_DATE
        AND a.status IN ('scheduled', 'in_progress')
        AND p_appointment_time BETWEEN a.appointment_time AND a.end_time;

    IF v_existing_count > 0 THEN
        RAISE EXCEPTION 'Time slot not available for this employee';
    END IF;

    -- Insert appointment
    INSERT INTO appointment (
        customer_id,
        employee_service_id,
        appointment_date,
        appointment_time,
        status
    ) VALUES (
        p_customer_id,
        p_employee_service_id,
        CURRENT_DATE,
        p_appointment_time,
        'scheduled'
    );

    RAISE NOTICE 'Appointment scheduled successfully for % at %', CURRENT_DATE, p_appointment_time;

EXCEPTION
    WHEN OTHERS THEN
        RAISE EXCEPTION 'Scheduling failed: %', SQLERRM;
END;
$$;
```

Figure 28 Stored Procedure 1 (Inserting an appointment)

The schedule_appointment procedure (Figure 28) is designed to streamline the process of scheduling an appointment in the database. It validates the inputs, checks for potential conflicts, and ensures data integrity before inserting a new appointment record. Here's a breakdown of its functionality:

Parameters

1. **p_customer_id (INT)**: The ID of the customer for whom the appointment is being scheduled.
2. **p_employee_service_id (INT)**: The ID representing the employee and service combination for the appointment.
3. **p_appointment_time (TIME)**: The desired appointment time.

Process Flow

1. **Retrieve Employee and Service IDs**:
    - Fetches the employee_id and service_id associated with the given p_employee_service_id from the employee_service table.
2. **Validation Steps**:
    - **Customer Check**: Ensures the provided p_customer_id exists in the customer table. If not, raises an exception: 'Invalid customer ID'.
    - **Employee Service Check**: Verifies that the retrieved employee_id and service_id are valid. If either is NULL, raises an exception: 'Invalid employee service combination'.
    - **Time Validation**: Checks that the provided p_appointment_time is not in the past. If it is, raises an exception: 'Cannot schedule appointments in the past'.
    - **Overlapping Appointment Check**: Ensures that the selected employee does not already have an overlapping appointment. It does this by:
        - Counting existing appointments for the employee that are scheduled or in progress.
        - Verifying if the p_appointment_time falls between the appointment_time and end_time of any existing appointments on the same day.

        - If overlaps exist, raises an exception: 'Time slot not available for this employee'.
3. **Insert Appointment**:
    - If all validations pass, inserts a new record into the appointment table with the following details:
        - customer_id: The provided p_customer_id.
        - employee_service_id: The provided p_employee_service_id.
        - appointment_date: The current date.
        - appointment_time: The provided p_appointment_time.
        - status: Set to 'scheduled'.
4. **Success Notification**:
    - If the appointment is successfully inserted, raises a notice: 'Appointment scheduled successfully for [DATE] at [TIME]'.

Exception Handling

- Captures any unexpected errors during execution using the EXCEPTION block.
- If an error occurs, raises an exception with a descriptive error message, including the SQL error message: 'Scheduling failed: [Error]'.

Key Benefits

- **Automated Validation**: Ensures all necessary checks are performed before scheduling an appointment.
- **Data Integrity**: Prevents conflicts such as overlapping appointments or invalid data entries.
- **Efficiency**: Simplifies the scheduling process by consolidating multiple validation steps into a single procedure.
- **Error Handling**: Provides clear error messages to identify issues during scheduling.

This procedure ensures that only valid and conflict-free appointments are created in the system, thereby improving operational efficiency and maintaining data accuracy.

2. Procedure: ‘update_appointment_status’

```
CREATE OR REPLACE PROCEDURE update_appointment_status(
    p_appointment_id INT,
    p_new_status appointment_status
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_current_status appointment_status;
BEGIN
    -- Get current status
    SELECT status INTO v_current_status
    FROM appointment
    WHERE id = p_appointment_id;

    -- Check if appointment exists
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Appointment with ID % not found', p_appointment_id;
    END IF;

    -- Validate status transitions
    CASE v_current_status
        WHEN 'scheduled' THEN
            IF p_new_status NOT IN ('in_progress', 'cancelled') THEN
                RAISE EXCEPTION 'Scheduled appointments can only be changed to in_progress or cancelled';
            END IF;
        WHEN 'in_progress' THEN
            IF p_new_status != 'completed' THEN
                RAISE EXCEPTION 'In-progress appointments can only be completed';
            END IF;
        WHEN 'completed' THEN
            RAISE EXCEPTION 'Cannot change status of completed appointments';
        WHEN 'cancelled' THEN
            RAISE EXCEPTION 'Cannot change status of cancelled appointments';
    END CASE;

    -- Update appointment status
    UPDATE appointment
    SET status = p_new_status
    WHERE id = p_appointment_id;

    RAISE NOTICE 'Appointment status updated to %', p_new_status;

EXCEPTION
    WHEN OTHERS THEN
        RAISE EXCEPTION 'Status update failed: %', SQLERRM;
END;
$$;
```

Figure 29 Stored procedure 2 (Update the status of appointment)

The update_appointment_status procedure (Figure 29) is designed to update the status of an appointment while ensuring the validity of the transition between statuses. It incorporates validation checks to maintain the integrity of appointment workflows and prevents invalid or undesired status changes.

Parameters

1. **p_appointment_id (INT)**: The ID of the appointment to be updated.
2. **p_new_status (appointment_status)**: The new status to which the appointment should be updated. The appointment_status is a predefined enum-like type representing valid statuses (e.g., scheduled, in_progress, completed, cancelled).

Process Flow

1. **Retrieve Current Status**:
    - Fetches the current status of the appointment identified by p_appointment_id from the appointment table and stores it in the v_current_status variable.
    - If no record is found for the given p_appointment_id, raises an exception: 'Appointment with ID % not found'.
2. **Status Transition Validation**:
    - Uses a CASE statement to validate the transition between the current status (v_current_status) and the new status (p_new_status). Specific rules are enforced:
        - **From scheduled**:
            - Can only transition to in_progress or cancelled. Any other transition raises an exception: 'Scheduled appointments can only be changed to in_progress or cancelled'.
        - **From in_progress**:
            - Can only transition to completed. Any other transition raises an exception: 'In-progress appointments can only be completed'.
        - **From completed**:
            - Status cannot be changed once completed. Raises an exception: 'Cannot change status of completed appointments'.
        - **From cancelled**:

        - Status cannot be changed once cancelled. Raises an exception: 'Cannot change status of cancelled appointments'.

3. **Update Appointment Status**:
    - If the validation checks pass, updates the status column in the appointment table for the record with ID p_appointment_id to the new status p_new_status.
4. **Success Notification**:
    - If the status update is successful, raises a notice: 'Appointment status updated to [NEW_STATUS]'.
5. **Exception Handling**:
    - Captures unexpected errors during execution and raises an exception with a detailed error message: 'Status update failed: [Error]'.

Key Benefits

1. **Controlled Status Transitions**:
    - Ensures that appointments follow logical status changes, such as moving from scheduled to in_progress or from in_progress to completed.
    - Prevents invalid transitions, like updating a completed or cancelled appointment.
2. **Data Integrity**:
    - By enforcing valid transitions, the procedure ensures that the appointment lifecycle remains consistent and logical.
3. **Error Transparency**:
    - Provides clear and detailed error messages when transitions fail, making debugging and user feedback easier.
4. **Simplification**:
    - Centralizes the logic for updating appointment statuses, making it easier to manage and extend in the future.

This procedure ensures that appointments transition between statuses in a controlled manner, preventing data inconsistencies and maintaining the integrity of the system's workflow. It is a critical component for managing appointment states in a structured and predictable way.

3. Procedure: ‘add_customer’

```
CREATE OR REPLACE PROCEDURE add_customer(
    p_first_name VARCHAR(50),
    p_last_name VARCHAR(50),
    p_gender gender_type,
    p_phone VARCHAR(15),
    p_email VARCHAR(100)
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Check if phone already exists
    IF EXISTS (SELECT 1 FROM customer WHERE phone_number = p_phone) THEN
        RAISE EXCEPTION 'Customer with phone % already exists', p_phone;
    END IF;

    -- Insert new customer
    INSERT INTO customer (
        first_name,
        last_name,
        gender,
        phone_number,
        email
    ) VALUES (
        p_first_name,
        p_last_name,
        p_gender,
        p_phone,
        p_email
    );

    RAISE NOTICE 'Customer % % added successfully', p_first_name, p_last_name;
EXCEPTION
    WHEN OTHERS THEN
        RAISE EXCEPTION 'Failed to add customer: %', SQLERRM;
END;
$$;
```

Figure 30 Stored procedure 3 (Add Customer)

The add_customer procedure (Figure 30) is designed to insert a new customer into the database while ensuring data integrity and avoiding duplication of customer records. It incorporates validation checks to prevent adding customers with duplicate phone numbers, which serve as a unique identifier.

Parameters

1. **p_first_name (VARCHAR(50))**: The first name of the customer.
2. **p_last_name (VARCHAR(50))**: The last name of the customer.
3. **p_gender (gender_type)**: The gender of the customer. This refers to a predefined type (gender_type) for ensuring valid and consistent gender entries.
4. **p_phone (VARCHAR(15))**: The phone number of the customer, which must be unique in the customer table.
5. **p_email (VARCHAR(100))**: The email address of the customer.

Process Flow

1. Phone Number Validation:

1. The procedure first checks if a customer with the given phone number (p_phone) already exists in the customer table.
2. If a match is found, it raises an exception:
    - 'Customer with phone % already exists', where % is replaced with the duplicate phone number.
3. This ensures that no duplicate records are inserted.

2. Insert New Customer:

1. If the phone number check passes, the procedure inserts the provided customer details into the customer table:
    - Columns inserted include first_name, last_name, gender, phone_number, and email.
2. The new customer record is created using the values provided as parameters.

3. Success Notification:

1. After successfully inserting the customer, the procedure raises a notice:
    - 'Customer % % added successfully', where % % is replaced with the customer's first and last names.
2. This message confirms successful execution to the user or developer.

4. Exception Handling:

1. If an error occurs during the procedure execution, it is captured in the EXCEPTION block.
2. A new exception is raised with a detailed error message:
    - 'Failed to add customer: %', where % is replaced by the system-generated error message (SQLERRM).

Key Benefits

1. Data Integrity:

- By ensuring that the phone number is unique, the procedure maintains the integrity of the customer table and avoids duplicate entries.

2. Validation:

- Built-in checks prevent invalid data from being added, reducing errors and inconsistencies.

3. Simplification:

- Encapsulates the logic for adding a customer into a single reusable procedure, reducing redundancy and improving code maintainability.

This procedure simplifies the process of adding customers, ensuring that the database remains clean, consistent, and free of duplicate entries. It is a critical component for managing customer records efficiently and reliably.

Conclusion

These stored procedures play a crucial role in simplifying system operations by automating complex processes and enforcing data consistency. They ensure that business rules are applied consistently, reduce redundancy, and improve system maintainability. Overall, they are an integral part of the system's architecture, contributing to its reliability and scalability.

TRANSACTIONS

Introduction

In this section, I explore PostgreSQL's built-in transaction management system and how it ensures data integrity and consistency. PostgreSQL, by default, executes each SQL statement within an implicit transaction, automatically committing it upon success or rolling it back in case of failure. Through this project, I have learned to leverage explicit transaction controls using BEGIN, COMMIT, ROLLBACK to handle complex multi-step operations. This has allowed me to group related operations into atomic units, ensuring either all steps succeed or the database reverts to its original state in case of an error.

Transaction 1-> Procedure: ‘set_employee_off_duty’

```
--off_duty TRASACTION HANDLING Procedure
CREATE OR REPLACE PROCEDURE set_employee_off_duty(
    p_employee_id INT
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_exists BOOLEAN;
    v_has_appointments BOOLEAN;
BEGIN
    -- First check if employee exists
    SELECT EXISTS (
        SELECT 1 FROM employee WHERE id = p_employee_id
    ) INTO v_exists;

    IF NOT v_exists THEN
        ROLLBACK;
        RAISE NOTICE 'Employee with ID % does not exist', p_employee_id;
        RETURN;
    END IF;

    -- Then check for appointments
    SELECT EXISTS (
        SELECT 1 FROM appointment a
        JOIN employee_service es ON a.employee_service_id = es.id
        WHERE es.employee_id = p_employee_id
        AND a.status IN ('scheduled', 'in_progress')
    ) INTO v_has_appointments;

    IF v_has_appointments THEN
        ROLLBACK;
        RAISE NOTICE 'Employee % has pending appointments', p_employee_id;
        RETURN;
    END IF;

    -- If all checks pass, update status
    UPDATE employee
    SET current_status = 'off_duty'
    WHERE id = p_employee_id;

    COMMIT;
    RAISE NOTICE 'Employee % set to off-duty successfully', p_employee_id;
END;
$$;
```

Figure 31 Transaction 1 (Update employee status)

The set_employee_off_duty (Figure 31) procedure is designed to handle transaction-based updates to an employee's status in the database. It ensures that the employee exists and does not have pending appointments before setting their status to "off_duty." This procedure demonstrates the use of transaction management techniques, including **commit** and **rollback**, to maintain data integrity.

Parameters:

- **p_employee_id (INT)**: The ID of the employee whose status is to be updated.

Process Flow:

1. **Check if the Employee Exists**:
    - The procedure first checks whether an employee with the provided ID exists in the employee table.
    - If no such employee is found, the transaction is rolled back, and a notice is raised: Employee with ID % does not exist.
2. **Check for Pending Appointments**:
    - Next, it checks whether the employee has any pending appointments that are either "scheduled" or "in_progress."
    - This is done by joining the appointment table with the employee_service table to identify linked appointments.
    - If pending appointments are found, the transaction is rolled back, and a notice is raised: Employee % has pending appointments.
3. **Update Employee Status**:
    - If the employee exists and has no pending appointments, the procedure proceeds to update the employee's current_status in the employee table to 'off_duty'.
4. **Commit the Transaction**:
    - After successfully updating the employee's status, the transaction is committed to make the changes permanent.
    - A success message is raised: Employee % set to off-duty successfully.
5. **Transaction Handling**:

- The procedure uses **ROLLBACK** to undo changes if any of the validation checks fail.
- A **COMMIT** ensures that the changes are saved when all checks pass.

Key Benefits:

1. **Data Integrity**:
   - Ensures that only valid employees with no pending appointments can be set to "off_duty."
   - Prevents the possibility of employees being marked as off-duty while they have active responsibilities.
2. **Transaction Management**:
   - the use of transactions to handle updates, using **rollback** for error cases and **commit** for successful operations.
3. **Error Handling**:
   - Provides clear and meaningful notices when operations fail due to invalid employee IDs or pending appointments.

This procedure not only highlights the importance of transaction handling in database operations but also ensures that the business rules for employee status management are adhered to strictly.

Transaction 2-> Procedure: ‘transfer_services Procedure’

```
CREATE OR REPLACE PROCEDURE transfer_services(
    p_from_employee_id INT,
    p_to_employee_id INT,
    p_service_id INT
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_old_employee_service_id INT;
    v_new_employee_service_id INT;
BEGIN
    -- Check if source employee exists and has the service
    SELECT id INTO v_old_employee_service_id
    FROM employee_service
    WHERE employee_id = p_from_employee_id
    AND service_id = p_service_id;

    IF NOT FOUND THEN
        RAISE NOTICE 'Source employee does not provide this service';
        RETURN;
    END IF;

    -- Check if target employee exists
    IF NOT EXISTS (SELECT 1 FROM employee WHERE id = p_to_employee_id) THEN
        ROLLBACK;
        RAISE NOTICE 'Target employee not found';
        RETURN;
    END IF;

    -- Assign service to target employee
    INSERT INTO employee_service (employee_id, service_id)
    VALUES (p_to_employee_id, p_service_id)
    RETURNING id INTO v_new_employee_service_id;

    -- Update appointments to reference the new employee_service_id
    UPDATE appointment
    SET employee_service_id = v_new_employee_service_id
    WHERE employee_service_id = v_old_employee_service_id;

    -- Remove service from source employee
    DELETE FROM employee_service
    WHERE id = v_old_employee_service_id;

    COMMIT;
    RAISE NOTICE 'Service transferred successfully, and appointments updated';

END;
$$;
```

Figure 32 Transaction 2 (Service transferring procedure)

The transfer_services procedure (Figure 32) is designed to transfer a specific service from one employee to another while ensuring that any related appointments are updated accordingly. It automates the transfer process and maintains data integrity.

Parameters

1. **p_from_employee_id (INT)**: The ID of the employee currently providing the service.
2. **p_to_employee_id (INT)**: The ID of the employee to whom the service is being transferred.
3. **p_service_id (INT)**: The ID of the service being transferred.

Process Flow

1. **Check if Source Employee Provides the Service**:
    - Retrieves the id of the record in the employee_service table where the source employee (p_from_employee_id) is associated with the specified service (p_service_id).
    - If no such record exists:
        - A notice is raised: 'Source employee does not provide this service'.
        - The procedure exits without making any changes.
2. **Validate Target Employee Existence**:
    - Checks if the target employee (p_to_employee_id) exists in the employee table.
    - If the target employee does not exist:
        - The transaction is rolled back using ROLLBACK.
        - A notice is raised: 'Target employee not found'.
        - The procedure exits.
3. **Assign Service to Target Employee**:
    - Inserts a new record into the employee_service table for the target employee and the specified service.
    - The id of the newly inserted record is stored in the variable v_new_employee_service_id.
4. **Update Related Appointments**:

- Updates all appointments associated with the old id (stored in v_old_employee_service_id) to reference the new id (stored in v_new_employee_service_id).

5. **Remove Service from Source Employee**:
    - Deletes the record from the employee_service table where the source employee is associated with the specified service (v_old_employee_service_id).
6. **Finalize Changes**:
    - A notice is raised: 'Service transferred successfully, and appointments updated' .

Error Handling

- **Employee Not Found**:
    - If the target employee does not exist, the transaction is rolled back, and a notice is raised.
- **No Service Found for Source Employee**:
    - If the source employee does not provide the specified service, the procedure exits without making any changes.

Key Benefits

1. **Data Integrity**:
    - Ensures all appointments are updated to reflect the new service provider.
    - Prevents orphaned records by removing the old employee_service record only after successful transfer.
2. **Efficiency**:
    - Automates the process of transferring services, avoiding manual updates to multiple tables.
3. **Transaction Management**:
    - Uses ROLLBACK and COMMIT to ensure atomicity, so either all changes are applied, or none are made if an error occurs.

This procedure is an efficient way to manage service reassignments within the system while ensuring data consistency and logical accuracy across related tables. It is a critical component for handling dynamic service assignments.

Conclusion

Understanding and utilizing PostgreSQL's transaction handling capabilities has significantly enhanced the reliability and robustness of the salon management system. By explicitly managing transactions in procedures and functions, I ensured data consistency and minimized errors, even during complex operations. The ability to use rollback and savepoints has provided fine-grained control over database changes, making the system more resilient and dependable. Overall, incorporating these practices has greatly improved the efficiency and integrity of the system.

CONCLUSION

This report highlights the design and implementation of a robust database system for a salon management system. It incorporates advanced features like stored procedures, triggers, and transaction handling to ensure automation, efficiency, and data integrity. By utilizing PostgreSQL's powerful capabilities, such as ACID-compliant transaction management and dynamic triggers, the system minimizes manual interventions and reduces the risk of errors.

The use of stored procedures streamlines critical operations, such as scheduling appointments, updating statuses, and managing employee availability. Triggers automate repetitive tasks, ensuring consistency across the database. Additionally, implementing proper transaction handling has provided a safeguard against partial updates, preserving the reliability of the system even during unforeseen issues.

Overall, this report demonstrates a comprehensive approach to building a scalable and maintainable database solution, showcasing the importance of well-structured queries, procedures, and transaction management in modern database systems.

REFERENCES

[1] "PostgreSQL Documentation," [Online]. Available: https://www.postgresql.org/docs/current/plpgsql-transactions.html.

[2] "W3 School PostgreSQL TUtorial," [Online]. Available: https://www.w3schools.com/postgresql/.