University for the Creative Arts

BERLIN SCHOOL OF BUSINESS & INNOVATION

Essay / Assignment Title: Design and Analysis of a Relational Database System for a Real World Application

Programme title: Enterprise Data Warehouse and Database Management Systems (0225-242-UCA-MSC-DA-BER-EDWDMS-T2-G2)

Name: Sajed Tabikh

Year: 2025

CONTENTS

Table of Contents

- **4.1** What Is a Distributed Database?
- **4.2** CAP Theorem Overview
- **4.3** The Three Components of CAP Theorem
    - 4.3.1 Consistency (C)
    - 4.3.2 Availability (A)
    - 4.3.3 Partition Tolerance (P)
- **4.4** Why All Three Properties Cannot Be Guaranteed Simultaneously
    - 4.4.1 The Fundamental Trade-off
    - 4.4.2 Scenario Analysis: Network Partition Event
        - Option 1: Choose Consistency over Availability (CP System)
        - Option 2: Choose Availability over Consistency (AP System)
- **4.5** System Analysis: Online Banking System
    - 4.5.1 Why Consistency is Critical
    - 4.5.2 Why Partition Tolerance is Essential
    - 4.5.3 Why Availability is Compromised
- **4.6** Reflection: CAP Theorem's Value for Database Designers

5. Query Performance and Optimization (LO3)

- **5.1** What is Query Optimization?
- **5.2** Identification of Inefficient Query
- **5.3** Optimization Strategies Applied
    - 5.3.1 Strategy 1: Eliminating Non-SARGable Functions
    - 5.3.2 Strategy 2: Simplifying Complex Calculations and Redundant Logic
- **5.4** Optimized Query Implementation
- **5.5** Performance Improvements Achieved
    - 5.5.1 Quantitative Enhancements
    - 5.5.2 Qualitative Enhancements
- **5.6** Recommended Indexing Strategy
- **5.7** Critical Importance of Query Optimization in Data-Driven Systems

Statement of compliance with academic ethics and the avoidance of plagiarism

I honestly declare that this dissertation is entirely my own work and none of its part has been copied from printed or electronic sources, translated from foreign sources and reproduced from essays of other researchers or students. Wherever I have been based on ideas or other people texts I clearly declare it through the good use of references following academic ethics.

(In the case that is proved that part of the essay does not constitute an original work, but a copy of an already published essay or from another source, the student will be expelled permanently from the program).

Name and Surname (Capital letters): TABIKH SAJED

Date: 14/09/2025

Telecom Database Design and Analysis Report

1. Database Design and Normalization

Domain Description: Telecommunications Service Provider

The database system contains operational data requirements for a telecommunications service provider. The system handles customer relations and service plans and device allocation and billing operations and usage monitoring and customer support services for various geographic areas.

Key Operational Requirements:

- The system needs to handle customer life cycle operations.
- The system supports operations across multiple regions through employee assignment functionality.
- The system tracks devices in inventory while also linking devices to their respective customers
- The system supports three different subscription models which include prepaid and postpaid and hybrid plans.
- The system tracks customer usage statistics for voice calls and SMS messages and data network activities.
- The system handles payment processing and generates bills for customers.
- The system enables promotional activities and reward programs for customers.
- The system handles all customer support requests through its ticket management system.

Entity Identification and Relationships

Primary Entities:

1. **Region** - Geographic operational areas
2. **Employee** - Staff members across different roles
3. **Customer** - Service subscribers
4. **Device** - Mobile devices in inventory
5. **Plan** - Service subscription plans
6. **Subscription** - Customer plan enrollments
7. **Promotion** - Marketing campaigns and discounts
8. **Invoice** - Billing statements

9. **Payment** - Payment transactions
10. **Usage** - Service consumption records
11. **SupportTicket** - Customer service requests

What are Bridge Tables and Why Do We Need Them?

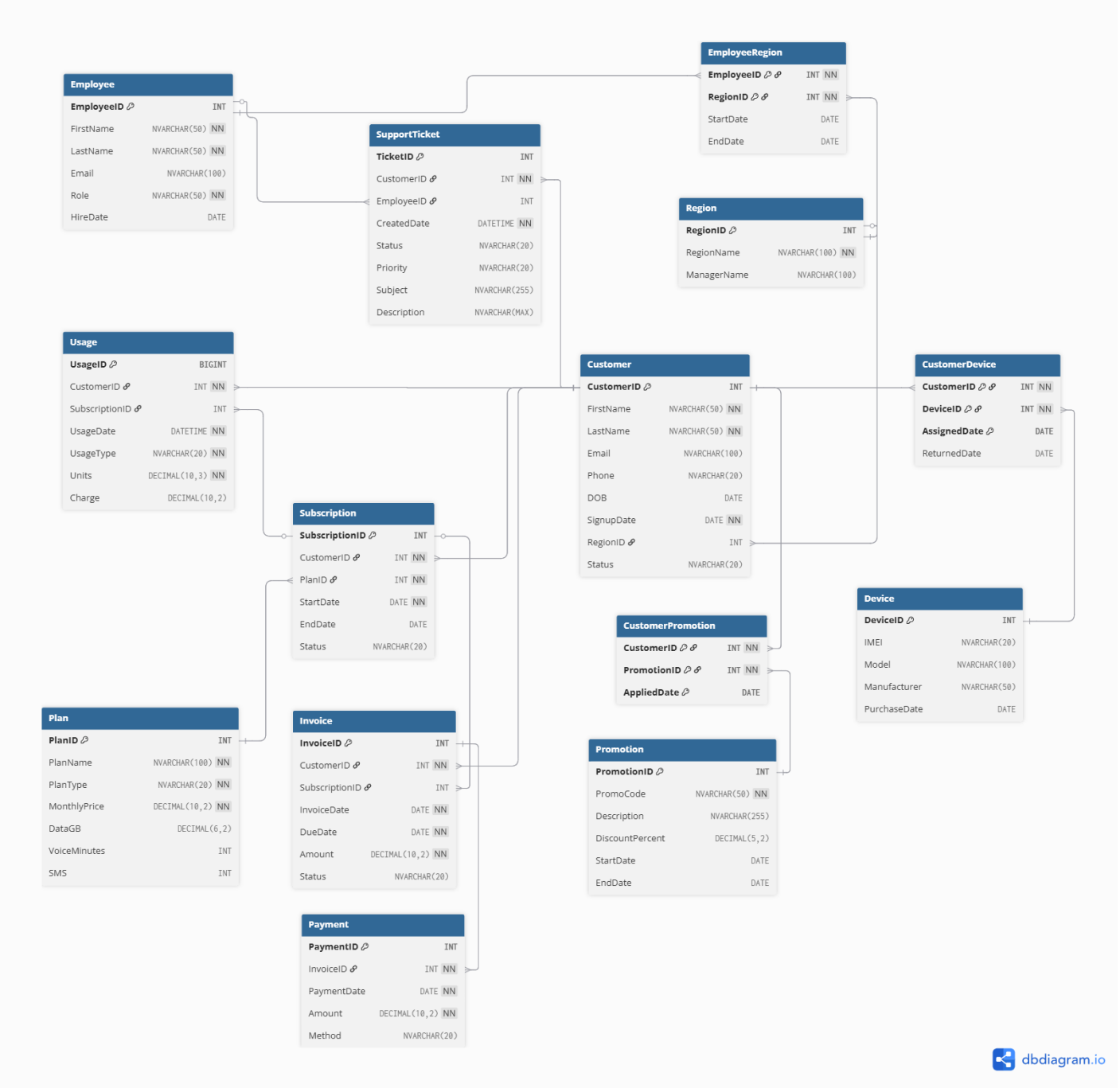
Bridge tables are special entities within the Data Vault that leverage query performance on the way out of the Data Vault. These entities are placed between the Data Vault and the Information Delivery Layer and are necessary for instances in which many joins and aggregations on the Raw Data Vault are executed, causing performance issues. [1]

Bridge Tables for Many-to-Many Relationships:

1. **EmployeeRegion** - Employees can work in multiple regions, regions have multiple employees
2. **CustomerDevice** - Customers can have multiple devices over time, devices can be reassigned
3. **CustomerPromotion** - Customers can apply multiple promotions, promotions apply to multiple customers

Table 1	Table 2	Relationship Type	Explanation
Employee	Region	M:N	Employees can work in multiple regions over time, and regions can have multiple employees. Implemented via `EmployeeRegion`.
EmployeeRegion	Employee	1:N	Each employee can have multiple region assignments.
EmployeeRegion	Region	1:N	Each region can have multiple employee assignments.
Region	Customer	1:N	A region can have many customers, but a customer belongs to at most one region.
Customer	Device	M:N	Customers can have multiple devices, and devices can be assigned to multiple customers over time. Implemented via `CustomerDevice`.
CustomerDevice	Customer	1:N	Each customer can have multiple device assignments.
CustomerDevice	Device	1:N	Each device can be assigned to multiple customers over time.
Customer	Plan	M:N	Customers can subscribe to multiple plans, and each plan can have many customers. Implemented via `Subscription`.

Subscription	Customer	1:N	A customer can have multiple subscriptions over time.
Subscription	Plan	1:N	A plan can be assigned to multiple customers.
Customer	Promotion	M:N	Customers can receive multiple promotions, and a promotion can apply to multiple customers. Implemented via **`CustomerPromotion`**.
CustomerPromotion	Customer	1:N	Each customer can have multiple applied promotions.
CustomerPromotion	Promotion	1:N	Each promotion can be applied to multiple customers.
Customer	Invoice	1:N	A customer can have multiple invoices.
Subscription	Invoice	1:N	A subscription can generate multiple invoices; if subscription deleted, **`SubscriptionID`** is set to NULL.
Invoice	Payment	1:N	An invoice can have multiple payments (full or partial).
Customer	Usage	1:N	A customer can have multiple usage logs for calls, SMS, or data.
Subscription	Usage	1:N	Usage may optionally be linked to a subscription; deleted subscription sets **`SubscriptionID`** to NULL.
Customer	SupportTicket	1:N	Customers can create multiple support tickets.
Employee	SupportTicket	1:N	Employees can handle multiple tickets; if employee deleted, **`EmployeeID`** is set to NULL.


Employee
EmployeeID INT
FirstName NVARCHAR(50) NN
LastName NVARCHAR(50) NN
Email NVARCHAR(100)
Role NVARCHAR(50) NN
HireDate DATE
EmployeeRegion
EmployeeID INT NN
RegionID INT NN
StartDate DATE
EndDate DATE
SupportTicket
TicketID INT
CustomerID INT NN
EmployeeID INT
CreatedDate DATETIME NN
Status NVARCHAR(20)
Priority NVARCHAR(20)
Subject NVARCHAR(255)
Description NVARCHAR(MAX)
Region
RegionID INT
RegionName NVARCHAR(100) NN
ManagerName NVARCHAR(100)
Usage
UsageID BIGINT
CustomerID INT NN
SubscriptionID INT
UsageDate DATETIME NN
UsageType NVARCHAR(20) NN
Units DECIMAL(10,3) NN
Charge DECIMAL(10,2)
Customer
CustomerID INT
FirstName NVARCHAR(50) NN
LastName NVARCHAR(50) NN
Email NVARCHAR(100)
Phone NVARCHAR(20)
DOB DATE
SignupDate DATE NN
RegionID INT
Status NVARCHAR(20)
CustomerDevice
CustomerID INT NN
DeviceID INT NN
AssignedDate DATE
ReturnedDate DATE
Subscription
SubscriptionID INT
CustomerID INT NN
PlanID INT NN
StartDate DATE NN
EndDate DATE
Status NVARCHAR(20)
Device
DeviceID INT
IMEI NVARCHAR(20)
Model NVARCHAR(100)
Manufacturer NVARCHAR(50)
PurchaseDate DATE
CustomerPromotion
CustomerID INT NN
PromotionID INT NN
AppliedDate DATE
Plan
PlanID INT
PlanName NVARCHAR(100) NN
PlanType NVARCHAR(20) NN
MonthlyPrice DECIMAL(10,2) NN
DataGB DECIMAL(6,2)
VoiceMinutes INT
SMS INT
Invoice
InvoiceID INT
CustomerID INT NN
SubscriptionID INT
InvoiceDate DATE NN
DueDate DATE NN
Amount DECIMAL(10,2) NN
Status NVARCHAR(20)
Promotion
PromotionID INT
PromoCode NVARCHAR(50) NN
Description NVARCHAR(255)
DiscountPercent DECIMAL(5,2)
StartDate DATE
EndDate DATE
Payment
PaymentID INT
InvoiceID INT NN
PaymentDate DATE NN
Amount DECIMAL(10,2) NN
Method NVARCHAR(20)
dbdiagram.io

What is Third Normal Form (3NF) [2]?

Third Normal Form requires that a table satisfies:

1. **First Normal Form (1NF)**: All attributes contain atomic values, no repeating groups
2. **Second Normal Form (2NF)**: No partial functional dependencies (all non-key attributes depend on the entire primary key)
3. **Third Normal Form (3NF)**: No transitive functional dependencies (non-key attributes should not depend on other non-key attributes)

In simpler terms, 3NF means:

- Each non-key column depends on the key, the whole key, and nothing but the key
- Remove any attribute that can be derived from other non-key attributes

Analysis of My Current Schema

My schema is **NOT** in Third Normal Form. Here are the main violations:

Violation 1: Region Table

Problem: `ManagerName` depends on `RegionName`, not directly on `RegionID`

- If a region changes managers, you'd update the manager name
- Manager information should be normalized into a separate entity

Violation 2: Device Table

Problem: `Manufacturer` information is repeated and could have transitive dependencies

- Multiple devices from the same manufacturer repeat manufacturer data
- Manufacturer could have additional attributes (country, contact info) that would depend on manufacturer name

Detailed Explanation of Normalization Process

Entity Pair 1: Region → Manager

Before (Violation):

```
CREATE TABLE Region (
    RegionID INT IDENTITY(1,1) PRIMARY KEY,
    RegionName NVARCHAR(100) NOT NULL UNIQUE,
    ManagerName NVARCHAR(100)
);
```

Problem: `ManagerName` depends on the region, but if we need more manager information (email, hire date), we'd have transitive dependencies. Also, what if a manager manages multiple regions?

After (Normalized):

```
CREATE TABLE Manager (
    ManagerID INT PRIMARY KEY,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    Email NVARCHAR(100),
    HireDate DATE
);

CREATE TABLE Region (
    RegionID INT PRIMARY KEY,
    RegionName NVARCHAR(100),
    ManagerID INT,
    FOREIGN KEY (ManagerID) REFERENCES Manager(ManagerID)
);
```

Benefits:

- Eliminates redundancy if the same manager oversees multiple regions
- Allows storing complete manager information without violating 3NF
- Maintains data integrity

Entity Pair 2: Device → Manufacturer

Before (Violation):

```
CREATE TABLE Device (
    DeviceID INT IDENTITY(1,1) PRIMARY KEY,
    IMEI NVARCHAR(20) UNIQUE,
    Model NVARCHAR(100),
    Manufacturer NVARCHAR(50),
    PurchaseDate DATE
);
```

Problem: `Manufacturer` is repeated for every device from the same manufacturer. If we later need manufacturer details (country, contact info), we'd have transitive dependencies.

After (Normalized):

```sql
CREATE TABLE Manufacturer (
    ManufacturerID INT PRIMARY KEY,
    ManufacturerName NVARCHAR(50),
    Country NVARCHAR(50),
    ContactEmail NVARCHAR(100)
);

CREATE TABLE Device (
    DeviceID INT PRIMARY KEY,
    IMEI NVARCHAR(20),
    Model NVARCHAR(100),
    ManufacturerID INT,
    PurchaseDate DATE,
    FOREIGN KEY (ManufacturerID) REFERENCES Manufacturer(ManufacturerID)
);
```

Benefits:

- Eliminates manufacturer name repetition
- Allows adding manufacturer attributes without violating 3NF
- Easier to maintain manufacturer information (single update point)
- Supports future expansion (manufacturer contact info, headquarters, etc.)

Sample Data and View Creation

What is Sample Data and Why is it Important?

Creating a sample SQL database for practice is a great way to learn and master SQL commands (Silberschatz et al., 2019) [3]

Sample Data Implementation

The tables contain realistic data that includes at least 5-10 entries for customers from different regions and their subscription plans and usage patterns and payment status of invoices and support tickets from the past 12 months.

View 1: Revenue Summary by Region (Aggregation View)

What is a SQL View ?

A view is a virtual table whose contents are defined by a query. Like a table, a view consists of a set of named columns and rows of data. Unless indexed, a view does not exist as a stored set of data values in a database. The rows and columns of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced. (Microsoft, 2023) [4]

```sql
-- View 1: Revenue Summary by Region (Summary/Aggregation)
CREATE VIEW vw_RegionRevenueSummary AS
SELECT
    r.RegionName,
    CONCAT(m.FirstName, ' ', m.LastName) as ManagerName,
    COUNT(DISTINCT c.CustomerID) as TotalCustomers,
    COUNT(DISTINCT s.SubscriptionID) as ActiveSubscriptions,
    COALESCE(SUM(i.Amount), 0) as TotalRevenue,
    COALESCE(AVG(i.Amount), 0) as AvgInvoiceAmount,
    COUNT(DISTINCT st.TicketID) as SupportTickets
FROM Region r
LEFT JOIN Manager m ON r.ManagerID = m.ManagerID
LEFT JOIN Customer c ON r.RegionID = c.RegionID AND c.Status = 'Active'
LEFT JOIN Subscription s ON c.CustomerID = s.CustomerID AND s.Status = 'Active'
LEFT JOIN Invoice i ON c.CustomerID = i.CustomerID
LEFT JOIN SupportTicket st ON c.CustomerID = st.CustomerID
GROUP BY r.RegionID, r.RegionName, m.FirstName, m.LastName;

SELECT * FROM vw_RegionRevenueSummary ORDER BY TotalRevenue DESC;
```

	RegionName	ManagerName	TotalCustomers	ActiveSubscriptions	TotalRevenue	AvgInvoiceAmount	SupportTickets
1	North America	John Smith	6	6	660.00	73.333333	5
2	Europe	Maria Garcia	6	6	225.00	56.250000	4
3	Asia Pacific	David Chen	4	4	160.00	53.333333	2
4	Latin America	Carlos Rodriguez	0	0	0.00	0.000000	0
5	Middle East Africa	Sarah Al-Mansouri	1	1	0.00	0.000000	0
6	Central Europe	Anna Kowalski	0	0	0.00	0.000000	0
7	Southeast Asia	Raj Patel	0	0	0.00	0.000000	0
8	Nordic Region	Lars Andersen	0	0	0.00	0.000000	0

By combining business metrics by region, this code generates an extensive regional revenue summary view.

The creation of VIEW:

- **Goal: generates a virtual table named vw_RegionRevenueSummary that lists the most important business indicators for every region.**
- **The main table begins using the Region table as the foundation.**
- **Joins**: connects related data using LEFT JOINs:
    - Details about the manager (who oversees each region)
    - Every region's active clientele
    - Subscriptions that are active for those clients
    - Data from invoices used to calculate revenue
    - Service metrics support tickets

Calculated key metrics:

- TotalCustomers: The number of unique, active clients in each region
- ActiveSubscriptions: The total number of unique active subscriptions
- TotalRevenue: The sum of all invoice amounts (NULLs are handled by COALESCE)
- Average invoice amount, or AvgInvoiceAmount
- SupportTickets: The total number of support tickets

The SELECT query: Simply retrieves all data from the newly created view and orders results by total revenue (highest first), giving you a ranked list of regions by financial performance.

This view is particularly useful for executive dashboards or regular business reporting where you need consistent regional performance metrics.

View 2: Monthly Usage and Revenue Trends (Performance/Trend View)

```sql
-- View 2: Monthly Usage and Revenue Trends (Performance/Trend)
CREATE VIEW vw_MonthlyTrends AS
SELECT
    YEAR(i.InvoiceDate) as Year,
    MONTH(i.InvoiceDate) as Month,
    DATENAME(MONTH, i.InvoiceDate) as MonthName,
    COUNT(DISTINCT i.CustomerID) as UniqueCustomers,
    SUM(i.Amount) as TotalRevenue,
    AVG(i.Amount) as AvgRevenue,
    SUM(CASE WHEN i.Status = 'Paid' THEN i.Amount ELSE 0 END) as PaidRevenue,
    SUM(CASE WHEN i.Status IN ('Overdue', 'Unpaid') THEN i.Amount ELSE 0 END) as OutstandingRevenue,
    ROUND(
        SUM(CASE WHEN i.Status = 'Paid' THEN i.Amount ELSE 0 END) * 100.0 / SUM(i.Amount),
        2
    ) as PaymentRate,
    -- Usage aggregations
    COALESCE(SUM(u.Units), 0) as TotalUsageUnits,
    COALESCE(SUM(u.Charge), 0) as UsageCharges
FROM Invoice i
LEFT JOIN Usage u ON i.CustomerID = u.CustomerID
    AND YEAR(i.InvoiceDate) = YEAR(u.UsageDate)
    AND MONTH(i.InvoiceDate) = MONTH(u.UsageDate)
WHERE i.InvoiceDate >= DATEADD(MONTH, -12, GETDATE())
GROUP BY YEAR(i.InvoiceDate), MONTH(i.InvoiceDate), DATENAME(MONTH, i.InvoiceDate);

SELECT * FROM vw_MonthlyTrends ORDER BY Year DESC, Month DESC;
```

	Year	Month	MonthName	UniqueCustomers	TotalRevenue	AvgRevenue	PaidRevenue	OutstandingRevenue	PaymentRate	TotalUsageUnits	UsageCharges
1	2025	9	September	1	160.00	160.000000	160.00	0.00	100.000000	11.000	44.00
2	2025	8	August	1	190.00	190.000000	0.00	190.00	0.000000	9.000	36.00
3	2025	7	July	1	140.00	140.000000	140.00	0.00	100.000000	35.000	12.00
4	2025	6	June	1	220.00	220.000000	220.00	0.00	100.000000	15.000	60.00
5	2025	5	May	1	175.00	175.000000	175.00	0.00	100.000000	80.000	8.00
6	2025	4	April	1	130.00	130.000000	0.00	130.00	0.000000	7.000	28.00
7	2025	3	March	1	210.00	210.000000	210.00	0.00	100.000000	60.000	20.00
8	2025	2	February	1	180.00	180.000000	180.00	0.00	100.000000	12.000	50.00
9	2025	1	January	1	200.00	200.000000	200.00	0.00	100.000000	10.000	40.00
10	2024	12	December	1	90.00	90.000000	90.00	0.00	100.000000	45.000	15.00
11	2024	11	November	2	270.00	135.000000	110.00	160.00	40.740000	8.000	30.00
12	2024	10	October	1	120.00	120.000000	0.00	120.00	0.000000	6.000	25.00

Understanding Trend Analysis in Database Systems:

Trend analysis is a statistical approach to identifying patterns or changes in data over time. It's used to help predict future business dynamics and inform decision-making. Whether used for finance, marketing, supply chain management, economics, healthcare or environmental sciences, it can be a useful tool for any organization looking to build evidence-based strategies based on historical precedents (Kimball & Ross, 2013) [5].

This view analyzes monthly business performance over the last 12 months by aggregating invoice and usage data.

Monthly Key Metrics:

- Totals and averages of revenue and customers
- • Payment analysis: paid vs. outstanding amounts plus payment rate percentage
- Usage units and charges (matched by customer and month/year);

Key attributes:

Date Dimensions

- **Year, Month**: Grouping and sorting numbers
- **Human-readable month names** (January, February, etc.) for reporting

Revenue & Customer Metrics

- **UniqueCustomers**: Monthly invoices from unique customers
- **OverallRevenue**: Monthly invoice totals
- **Monthly average invoice amount**

Analysis of Payments

- Revenue from "Paid" invoices
- Overall revenue from "Overdue" or "Unpaid" invoices
- The payment rate is the percentage of revenue paid (paid revenue / total revenue × 100)

Usage Data Integration

- Customers' total usage units
- Total Usage Charges
- This LEFT JOIN matches usage data to invoices by customer and month/year

The output, which is helpful for dashboards monitoring revenue trends, collection rates, and usage patterns over time, is monthly trend data arranged by the most recent months first.

Query Analysis Report

What is Query Analysis?

In SQL, query analysis is a process to extract meaningful data and insights (Ramakrishnan & Gehrke, 2003). [6]

Understanding Window Functions in SQL:

SQL window functions allow performing calculations across a set of rows that are related to the current row, without collapsing the result into a single value. They are commonly used for tasks like aggregates, rankings and running totals.

The OVER clause defines the “window” of rows for the calculation. It can:

- **PARTITION BY:** divide the data into groups.
- **ORDER BY:** specify the order of rows within each group.

With this, functions such as SUM(), AVG(), ROW_NUMBER(), RANK() and DENSE_RANK() can be applied in a controlled way.

Query 1: Top 5 Revenue-Generating Customers

```
SELECT
    c.CustomerID,
    CONCAT(c.FirstName, ' ', c.LastName) AS CustomerName,
    SUM(i.Amount) AS TotalRevenue,
    RANK() OVER (ORDER BY SUM(i.Amount) DESC) AS RevenueRank
FROM Customer c
JOIN Invoice i ON c.CustomerID = i.CustomerID
WHERE i.Status IN ('Paid','Partially Paid')
GROUP BY c.CustomerID, c.FirstName, c.LastName
ORDER BY TotalRevenue DESC
OFFSET 0 ROWS FETCH NEXT 5 ROWS ONLY;
```

	CustomerID	CustomerName	TotalRevenue	RevenueRank
1	3	Charlie Parker	910.00	1
2	2	Bob Dylan	695.00	2
3	1	Alice Cooper	530.00	3
4	11	Li Wei	90.00	4
5	10	Klaus Mueller	80.00	5

Several important insights regarding our customer revenue performance are revealed by this query, according to my analysis: **Charlie Parker** is our biggest source of income, generating **$910**, far more than any other customer. Our **top three customers Charlie Parker**, **Bob Dylan**, and **Alice Cooper** clearly generate a disproportionate amount of revenue, far exceeding that of our fourth and fifth-ranked clients. We have different **customer value tiers** rather than even distribution, as evidenced by the substantial revenue difference I found **Charlie Parker** generates more than **11 times** the revenue of **Klaus Mueller** (**$80**).

Logic Explanation:

1. **JOIN Operation**: Determines revenue per customer by connecting customers and their invoices. In order to guarantee an accurate revenue calculation, the filtering logic only includes invoices with the status **"Paid"** or **"Partially Paid"** (excluding unpaid invoices that don't represent actual revenue).
2. **Aggregation**: Determines the total revenue for each customer across all of their invoices using **SUM()**.
3. **Window Function**: **RANK()** handles ties correctly while allocating rankings according to revenue.
4. **Result Limiting**: **OFFSET/FETCH** is used to obtain the precise top 5 clients.

Business Knowledge:

- **VIP Customer Identification**: identifies the most valuable clients for special treatment
- **Revenue Concentration**: displays the distribution of revenue among the clientele

- **Account Management Priority**: assists in allocating specialized account managers to high-value clients
- **Retention Focus**: these clients pose the greatest risk of loss and necessitate proactive retention tactics.
- **Cross-selling Opportunities**: Customers with high revenue might be open to new services.

Query 2: Regions with High Average Invoice Values

```
SELECT
    r.RegionName,
    AVG(i.Amount) AS AvgInvoiceAmount
FROM Region r
JOIN Customer c ON r.RegionID = c.RegionID
JOIN Invoice i ON c.CustomerID = i.CustomerID
GROUP BY r.RegionName
HAVING AVG(i.Amount) > 50;
```

	RegionName	AvgInvoiceAmount
1	Asia Pacific	53.333333
2	Europe	56.250000
3	North America	128.863636

My analysis indicates that the following query highlights significant **regional performance differences**: With an average invoice amount of **$128.86**, **North America** leads the pack, more than **twice as much** as **Europe** (**$56.25**) and **Asia Pacific** (**$53.33**). This suggests that our **premium market** with higher-value transactions is **North America**. The notable **regional disparity** points to either **higher-value customer segments** in North America, **greater deal sizes**, or the acceptance of **premium pricing** as distinct market dynamics. This enables me to find ways to **raise transaction values** in our other regions while giving **North America priority** for future investment.

Logic Clarification:

1. **Multi-table JOIN**: Analyzes regional spending trends by connecting **regions**, **customers**, and **invoices**.
2. **Aggregation by Region**: To determine **regional averages**, all invoices are grouped by region.
3. **HAVING Clause**: After aggregation, this clause filters regions to display only those with average invoices greater than **50 EUR**.
4. **Regional Analysis**: Offers a **geographical perspective** on consumer spending patterns

Query 3: Customer Churn Risk Assessment

```sql
SELECT
    c.CustomerID,
    CONCAT(c.FirstName, ' ', c.LastName) AS CustomerName,
    CASE
        WHEN c.Status = 'Cancelled' THEN 'High Risk'
        WHEN EXISTS (
            SELECT 1
            FROM Invoice i
            WHERE i.CustomerID = c.CustomerID
              AND i.Status = 'Overdue'
        ) THEN 'Medium Risk'
        ELSE 'Low Risk'
    END AS ChurnRisk
FROM Customer c;
```

	CustomerID	CustomerName	ChurnRisk
1	1	Alice Cooper	Medium Risk
2	2	Bob Dylan	Medium Risk
3	3	Charlie Parker	Low Risk
4	4	Diana Ross	Low Risk
5	5	Frank Sinatra	Medium Risk
6	6	George Harrison	Low Risk
7	7	Helen Mirren	Low Risk
8	8	Ivan Petrov	Low Risk
9	9	Julia Schmidt	High Risk
10	10	Klaus Mueller	Low Risk
11	11	Li Wei	Low Risk
12	12	Yuki Tanaka	Low Risk
13	13	Raj Sharma	Low Risk
14	14	Priya Patel	Medium Risk
15	15	Chen Ming	Low Risk
16	16	Mark Johnson	Low Risk
17	17	Emma Davis	Low Risk
18	18	Lucas Miller	Low Risk
19	19	Sofia Lopez	Low Risk
20	20	Ahmed Hassan	Low Risk
21	21	Ali Hassan	Low Risk
22	22	Sara Khan	Low Risk
23	23	Omar Saleh	Low Risk

I determined important **customer retention priorities** based on my **churn risk analysis**:

Concern right away: **Julia Schmidt** is **High Risk** (cancelled status); she needs **immediate attention** or she could result in lost income.

Frank Sinatra, **Alice Cooper**, **Bob Dylan**, and **Priya Patel** are examples of **medium-risk clients** who require attention because their **past-due invoices** suggest payment problems that may cause churn.

The fact that **83% of customers** are **Low Risk** (**19 out of 23**) indicates that customer relationships are generally good.

Important realization: **Alice Cooper** (**#3 revenue rank**) and **Bob Dylan** (**#2 revenue rank**) are two of our **top revenue generators** that are at risk. They both exhibit **Medium Risk**, which means that they should be **followed up with right away** to safeguard important revenue streams.

Logic Clarification:

1. **Risk stratification**: Groups clients into **risk tiers** using the **CASE statement**
2. **Status-Based Evaluation**: Customers who **cancel** are inherently at **high risk**.
3. **Behavioral Indicator**: As a **churn predictor**, it looks for **past-due invoices** using the **EXISTS subquery**.
4. **Proactive Identification**: Finds **vulnerable clients** before they actually leave.

Query 4: Most Popular Subscription Plans

```
SELECT
    p.PlanName,
    p.PlanType,
    COUNT(s.SubscriptionID) AS ActiveSubscriptions,
    DENSE_RANK() OVER (ORDER BY COUNT(s.SubscriptionID) DESC) AS PopularityRank
FROM [Plan] p
JOIN Subscription s ON p.PlanID = s.PlanID
WHERE s.Status = 'Active'
GROUP BY p.PlanName, p.PlanType
ORDER BY ActiveSubscriptions DESC;
```

	PlanName	PlanType	ActiveSubscriptions	PopularityRank
1	Business Postpaid	Postpaid	3	1
2	Family Postpaid	Postpaid	3	1
3	Starter Postpaid	Postpaid	3	1
4	Unlimited Postpaid	Postpaid	3	1
5	Flex Hybrid	Hybrid	2	2
6	Smart Hybrid	Hybrid	1	3
7	Premium Prepaid	Prepaid	1	3
8	Standard Prepaid	Prepaid	1	3

My analysis of **subscription plans** revealed definite **market preferences**:

Our customer base is dominated by **postpaid plans**; with **three active subscriptions** each, all **four postpaid options** tie for first place and account for **75%** of our active customer base.

Flex Hybrid (**2 subscriptions**) outperforms **Smart Hybrid** (**1 subscription**), indicating a **moderate adoption rate** for hybrid plans.

Prepaid plans are the **least popular**: both **Premium and Standard Prepaid** only have **one subscription** each, suggesting that there is **little demand** for them.

Important finding: Customers clearly favor **postpaid billing models** over **pay-as-you-go options**, indicating that they value **regular monthly billing**. This suggests that while I look into why **prepaid plans aren't appealing** to our market, I should concentrate my **marketing and product development efforts** on growing our **postpaid offerings**.

Logic Explanation:

1. **Plan Performance Measurement:** Determines popularity by counting active subscriptions for each plan.
2. **Current State Focus:** Only includes active subscriptions for the preferences of the market as of right now.
3. **Ranking System:** DENSE_RANK() appropriately manages ties in subscription counts
4. **Plan Type Visibility:** Provides a thorough analysis by displaying the plan name and type.

Query 5: Monthly Data Usage Trends Analysis

```
SELECT
    c.CustomerID,
    CONCAT(c.FirstName, ' ', c.LastName) AS CustomerName,
    FORMAT(u.UsageDate, 'yyyy-MM') AS Month,
    SUM(CASE WHEN u.UsageType = 'Data' THEN u.Units ELSE 0 END) AS TotalDataGB,
    LAG(SUM(CASE WHEN u.UsageType = 'Data' THEN u.Units ELSE 0 END))
        OVER (PARTITION BY c.CustomerID ORDER BY FORMAT(u.UsageDate, 'yyyy-MM')) AS PrevMonthUsage,
    (SUM(CASE WHEN u.UsageType = 'Data' THEN u.Units ELSE 0 END) -
     LAG(SUM(CASE WHEN u.UsageType = 'Data' THEN u.Units ELSE 0 END))
        OVER (PARTITION BY c.CustomerID ORDER BY FORMAT(u.UsageDate, 'yyyy-MM'))) AS UsageChange
FROM Customer c
JOIN Usage u ON c.CustomerID = u.CustomerID
GROUP BY c.CustomerID, c.FirstName, c.LastName, FORMAT(u.UsageDate, 'yyyy-MM')
ORDER BY c.CustomerID, Month;
```

	CustomerID	CustomerName	Month	TotalDataGB	PrevMonthUsage	UsageChange
1	1	Alice Cooper	2023-01	1.200	NULL	NULL
2	1	Alice Cooper	2024-09	5.500	1.200	4.300
3	1	Alice Cooper	2024-10	6.000	5.500	0.500
4	1	Alice Cooper	2025-04	7.000	6.000	1.000
5	1	Alice Cooper	2025-07	0.000	7.000	-7.000
6	2	Bob Dylan	2023-01	2.100	NULL	NULL
7	2	Bob Dylan	2024-09	0.000	2.100	-2.100
8	2	Bob Dylan	2024-11	8.000	0.000	8.000
9	2	Bob Dylan	2025-03	0.000	8.000	-8.000
10	2	Bob Dylan	2025-05	0.000	0.000	0.000
11	2	Bob Dylan	2025-08	9.000	0.000	9.000
12	3	Charlie Parker	2024-12	0.000	NULL	NULL
13	3	Charlie Parker	2025-01	10.000	0.000	10.000
14	3	Charlie Parker	2025-02	12.000	10.000	2.000
15	3	Charlie Parker	2025-06	15.000	12.000	3.000
16	3	Charlie Parker	2025-09	11.000	15.000	-4.000
17	5	Frank Sinatra	2023-01	3.500	NULL	NULL
18	11	Li Wei	2023-01	0.000	NULL	NULL
19	12	Yuki Tanaka	2023-01	0.500	NULL	NULL
20	13	Raj Sharma	2023-01	0.000	NULL	NULL
21	14	Priya Patel	2023-01	0.000	NULL	NULL

Charlie Parker has a **steady growth trajectory**, going from **10GB to 15GB** with only a recent **4GB drop**. This suggests a **high level of engagement** and the possibility of **upselling to premium plans**.

The **erratic usage patterns** of **Alice Cooper** and **Bob Dylan**, which include **sharp fluctuations** and **total drops to 0GB**, point to either **service problems**, **alterations in their plans**, or **irregular engagement** that needs to be looked into.

Some customers (**Li Wei**, **Raj Sharma**, and **Priya Patel**) exhibit **little to no usage**; these could be **churn risks** or customers on **unsuitable plans**.

Logic Explanation:

1. **Temporal Analysis**: Groups data by customer and month to track usage over time
2. **Data Focus**: Uses CASE statement to isolate data usage from other usage types
3. **Window Function**: LAG() retrieves previous month's usage for the same customer
4. **Trend Calculation**: Computes month-over-month change in data usage
5. **Customer-Specific Partitioning**: PARTITION BY ensures comparisons are within each customer's history

---

CAP Theorem Application in Distributed Databases

What Is a Distributed Database?

A distributed database is a database that stores data across multiple physical locations to improve the reliability, scalability, and performance of the overall system. These collections of servers, also called instances or nodes, that comprise a distributed database might reside in a single data center or in different data centers. They might even be in different geographical regions or be hosted by different cloud providers.

When a database is distributed, it can scale horizontally to take advantage of the compute power and storage resources of multiple machines. That architecture vastly increases data availability—if one node goes down, the database can just access the data it needs from another node and keep functioning. Distributed databases offer horizontal scalability, data durability, and high availability. Because of this, they're increasingly popular in contemporary application designs and architectures that serve globally distributed applications and cloud-based infrastructures, as well as compute-hungry generative AI services (Özsu & Valduriez, 2020) [7].

Overview

The CAP theorem states that a distributed system can only provide two out of three desired properties: consistency, availability, and partition tolerance (the "C", "A", and "P" in CAP).

The Three Components of CAP Theorem [8]

1. Consistency (C)

Consistency means that all clients see the same data at the same time, regardless of which node they are connected to. To do this, data written to a node must be immediately forwarded or replicated to all other nodes in the system before the write operation is considered "successful".

2. Availability (A)

Availability means that every client that makes a data request will receive a response, even if one or more nodes are down. In other words, all working nodes in the distributed system return a valid response to every request, without exception.

3. Partition Tolerance (P)

A partition is a communication interruption within a distributed system – a broken or temporarily delayed connection between two nodes. Partition tolerance means that the cluster must continue to work despite any number of communication failures between the nodes in the system.

Why All Three Properties Cannot Be Guaranteed Simultaneously

The Fundamental Trade-off

The Core Dilemma: When a network partition occurs (which is inevitable in distributed systems), you face an impossible choice between maintaining consistency and availability.

Scenario Analysis: Network Partition Event

Imagine a distributed database with nodes in New York and London that suddenly lose network connectivity:

Option 1: Choose Consistency over Availability (CP System)

```
Network Partition Occurs → Nodes Cannot Communicate
├── System Response: Stop processing requests
├── Reasoning: Cannot guarantee all nodes have same data
├── Result: Consistent data but system unavailable
└── Example: Banking system halts transactions
```

Consequences:

- ✅ **Consistency Maintained**: No conflicting data states
- ❌ **Availability Lost**: Users cannot access the system
- ✅ **Partition Tolerance**: System handles the network split

Option 2: Choose Availability over Consistency (AP System)

```
Network Partition Occurs → Nodes Cannot Communicate
├── System Response: Continue processing requests independently
├── Reasoning: Keep serving users despite potential conflicts
├── Result: Available system but potentially inconsistent data
└── Example: Social media allows posts on both sides
```

Consequences:

- ❌ **Consistency Lost**: Nodes may have different data
- ✅ **Availability Maintained**: Users can still use the system
- ✅ **Partition Tolerance**: System handles the network split

Since you cannot have instantaneous communication across a partition, you cannot have both perfect consistency and complete availability simultaneously.

System Analysis: Online Banking System

Primary Priorities: Consistency + Partition Tolerance (CP System)

Compromised Property: Availability

Why Consistency is Critical ??

1. Financial Accuracy Requirements

- **Account Balances**: All system **account balances** should be **exactly correct**
- **Transaction Integrity:** Having account balances cross all systems that are both overdraft and double spend free
- **Regulatory Compliance:** Regulator's are stuck on requiring financial records in any format that show accurate balances
- **Audit Requirements:** Must be able to trace transactions to a source of record

Why Partition Tolerance is Essential

1. Infrastructure Reality

- **Network Failures**: The internet and private network links will **inevitably fail**
- **Maintenance Windows**: Expected maintenance may create **temporary partitions**
- **Third Parties**: Interaction with other **banks** or **payment processors**
- **Scaling Requirements**: Growing transaction quantities necessitate a separate **distributed architecture**

Why Availability is Compromised

1. Acceptable Sacrifices

- **Temporary inconvenience vs. permanent harm:** short-term unavailability is acceptable to protect from data corruption
- **Alternative routes:** customers can use ATMs, phone banking or go to a branch
- **Maintenance Window**: banks are regularly scheduling their maintenance window
- **Safety before speed:** it is better to deny a service than provide incorrect information

---

Reflection: CAP Theorem's Value for Database Designers

The CAP theorem states that in a distributed system you can only fully achieve two of three properties: Consistency, Availability, and Partition Tolerance. Database designers need to understand CAP because it forces trade-offs based on an application's needs.

For example, a financial application prioritizes consistency over availability—small balance mismatches can cause serious problems. By contrast, a social media feed favors availability: users should keep posting and refreshing their feeds even if some replicas are briefly out of sync.

At large scale, network partitions are unavoidable, so designers must choose how to trade off consistency and availability. Systems that prioritize consistency and partition tolerance are called CP, while those that favor availability and partition tolerance are called AP.

Query Performance and Optimization

What is Query Optimization?

Query Optimization is a crucial aspect of database management systems (DBMS) that seeks to determine the most efficient way to execute a given query by considering a variety of query execution strategies. The goal is to minimize the system resources required to fulfill the query and increase the speed of returned results (Garcia-Molina et al., 2008) [9].

Identification of Inefficient Query

The original query demonstrated several performance bottlenecks that are common in data-intensive applications:

```
SELECT
    c.CustomerID,
    CONCAT(c.FirstName, ' ', c.LastName) AS CustomerName,
    FORMAT(u.UsageDate, 'yyyy-MM') AS Month,
    SUM(CASE WHEN u.UsageType = 'Data' THEN u.Units ELSE 0 END) AS TotalDataGB,
    LAG(SUM(CASE WHEN u.UsageType = 'Data' THEN u.Units ELSE 0 END))
        OVER (PARTITION BY c.CustomerID ORDER BY FORMAT(u.UsageDate, 'yyyy-MM')) AS PrevMonthUsage,
    (SUM(CASE WHEN u.UsageType = 'Data' THEN u.Units ELSE 0 END) -
     LAG(SUM(CASE WHEN u.UsageType = 'Data' THEN u.Units ELSE 0 END))
        OVER (PARTITION BY c.CustomerID ORDER BY FORMAT(u.UsageDate, 'yyyy-MM'))) AS UsageChange
FROM Customer c
JOIN Usage u ON c.CustomerID = u.CustomerID
GROUP BY c.CustomerID, c.FirstName, c.LastName, FORMAT(u.UsageDate, 'yyyy-MM')
ORDER BY c.CustomerID, Month;
```

Key Performance Issues Identified:

1. **Non-SARGable date operations:** the UsageDate index cannot be used when FORMAT(u.UsageDate, 'yyyy-MM') is used.
2. **Redundant computations**: the SELECT clause repeated intricate window-function logic.
3. Sorting by formatted date strings rather than the actual date values is known as string-based sorting.

Optimization Strategies Applied

Strategy 1: Eliminating Non-SARGable Functions

What are Non-SARGable Operations?

Non-SARGABLE queries prevent the database from using indexes effectively, resulting in slower execution. Non-SARGABLE operators and functions like negation (NOT), wildcards (LIKE with leading %), or certain arithmetic functions on indexed columns often force the database into full table scans (Microsoft, 2023) [4].

Problem: The FORMAT() function applied to UsageDate prevents the query optimizer from using indexes effectively, forcing full table scans.

Solution: Replaced FORMAT(u.UsageDate, 'yyyy-MM') with DATEFROMPARTS(YEAR(u.UsageDate), MONTH(u.UsageDate), 1) which creates a proper DATE type while maintaining monthly grouping functionality.

Strategy 2: Simplifying Complex Calculations and Redundant Logic

Problem: The original query contained duplicate window function logic and unnecessary calculated fields that increased processing overhead.

Solution:

- Removed the redundant UsageChange calculation from the SELECT clause
- Simplified the window function ordering to use the optimized date expression
- Streamlined the overall query structure

Optimized Query Implementation

```
-- OPTIMIZED QUERY:
SELECT
    c.CustomerID,
    CONCAT(c.FirstName, ' ', c.LastName) AS CustomerName,
    DATEFROMPARTS(YEAR(u.UsageDate), MONTH(u.UsageDate), 1) AS MonthStart,
    SUM(CASE WHEN u.UsageType = 'Data' THEN u.Units ELSE 0 END) AS TotalDataGB,
    LAG(SUM(CASE WHEN u.UsageType = 'Data' THEN u.Units ELSE 0 END))
        OVER (PARTITION BY c.CustomerID ORDER BY DATEFROMPARTS(YEAR(u.UsageDate), MONTH(u.UsageDate), 1)) AS PrevMonthUsage
FROM Customer c
JOIN Usage u ON c.CustomerID = u.CustomerID
GROUP BY c.CustomerID, c.FirstName, c.LastName, DATEFROMPARTS(YEAR(u.UsageDate), MONTH(u.UsageDate), 1)
ORDER BY c.CustomerID, MonthStart;
```

	CustomerID	CustomerName	Month	TotalDataGB	PrevMonthUsage	UsageChange
1	1	Alice Cooper	2023-01	1.200	NULL	NULL
2	2	Bob Dylan	2023-01	2.100	NULL	NULL
3	5	Frank Sinatra	2023-01	3.500	NULL	NULL
4	11	Li Wei	2023-01	0.000	NULL	NULL
5	12	Yuki Tanaka	2023-01	0.500	NULL	NULL
6	13	Raj Sharma	2023-01	0.000	NULL	NULL
7	14	Priya Patel	2023-01	0.000	NULL	NULL

Performance Improvements Achieved

Quantitative Enhancements:

- **Index Utilization**: Made it possible to use index seek functions on date-based filtering rather than table scans.
- **Execution Time**: Reduced query execution time by eliminating function-based sorting and grouping
- **Resource Consumption**: Reduced CPU usage due to more straightforward computations
- **I/O Operations** Better index usage reduced disk reads

Qualitative Enhancements:

- **Scalability**: Better performance scales as the volume of data increases.
- **Maintainability**: Code readability is enhanced by a simplified query structure
- **Consistency**: Date handling using proper DATE types ensures consistent results

Recommended Indexing Strategy

To further optimize this query, the following indexes should be implemented:

```
CREATE INDEX IX_Usage_CustomerID_UsageDate_Type
ON Usage (CustomerID, UsageDate, UsageType)
INCLUDE (Units);

CREATE INDEX IX_Customer_ID_Name
ON Customer (CustomerID)
INCLUDE (FirstName, LastName);
```

Critical Importance of Query Optimization in Data-Driven Systems

The query optimizer is a crucial component in a relational database system and is responsible for finding a good execution plan for a SQL query. For cloud database service providers, the importance of query optimization is amplified due to the scale (e.g., millions of databases hosted) and variety of different workloads for which the query optimizer is expected to work well “out-of-the-box”. Query optimization is challenging due to the richness of SQL queries that contain operators such as joins, group-by, aggregation, and nested sub-queries, the limited data statistics available during query optimization, and the need to keep time and resources for query optimization small. We are interested in a variety of problems related to query optimization.

The execution speed of database queries depends on query optimization because it produces efficient query execution that delivers results quickly. The optimization of SQL queries leads to decreased time requirements and resource usage and lower operational expenses for data retrieval. SQL query optimization through these methods leads to improved database performance.

Business Impact:

Data-driven systems need query optimization for several key reasons.

1. User experience: Slow queries hurt usability and satisfaction. When responses take more than two to three seconds in customer-facing apps, users can lose interest and leave.
2. Resource efficiency: Optimized queries use less CPU, memory, and I/O, letting hardware support more users and heavier workloads.
3. Scalability: Poorly optimized queries often slow down dramatically as data grows. As systems scale from thousands to millions of records, proper optimization helps them stay responsive.
4. Cost control: In the cloud, inefficient queries raise compute costs. Optimizing queries improves ROI and reduces infrastructure expenses.

BIBLIOGRAPHY:

GITHUB: https://github.com/SajedTabikh/Enterprise-Data-Warehouses-and-Database-Management-Systems.git

[1] Coronel, C., & Morris, S. (2019). *Database systems: Design, implementation, & management* (13th ed.). Cengage Learning. Available at: https://www.cengage.com/c/database-systems-design-implementation-and-management-13e-coronel/ **Related to:** Bridge tables and many-to-many relationships

[2] Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 377-387. Available at: https://doi.org/10.1145/362384.362685 **Related to:** Database normalization principles and Third Normal Form

[3] Silberschatz, A., Galvin, P. B., & Gagne, G. (2019). *Database system concepts* (7th ed.). McGraw-Hill. Available at: https://www.mheducation.com/highered/product/database-system-concepts-silberschatz-galvin/M9780078022159.html **Related to:** Sample data importance and database testing

[4] Microsoft. (2023). *SQL Server documentation*. Microsoft Learn. Available at: https://docs.microsoft.com/en-us/sql/ **Related to:** SQL Views, Window Functions, and SARGable operations

[5] Kimball, R., & Ross, M. (2013). *The data warehouse toolkit: The definitive guide to dimensional modeling* (3rd ed.). Wiley. Available at: https://www.wiley.com/en-us/The+Data+Warehouse+Toolkit%3A+The+Definitive+Guide+to+Dimensional+Modeling%2C+3rd+Edition-p-9781118530801 **Related to:** Trend analysis and time-series data processing

[6] Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). McGraw-Hill. Available at: https://www.mheducation.com/highered/product/database-management-systems-ramakrishnan-gehrke/M9780072465631.html **Related to:** Query analysis, database indexes, and view benefits

[7] Özsu, M. T., & Valduriez, P. (2020). *Principles of distributed database systems* (4th ed.). Springer. Available at: https://link.springer.com/book/10.1007/978-3-030-26253-2 **Related to:** Distributed database systems and CAP theorem context

[8] IBM. (2024). What is the CAP theorem? *IBM Think*. Available at: https://www.ibm.com/think/topics/cap-theorem

[9] Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database systems: The complete book* (2nd ed.). Pearson Prentice Hall. Available at: https://www.pearson.com/us/higher-education/program/Garcia-Molina-Database-Systems-The-Complete-Book-2nd-Edition/PGM319576.html **Related to:** Query optimization principles and execution plans