The Kreitzer Law Firm

Final Project Submission

Devan Kreitzer

ISM6218.003

Table of Contents

1. Executive Summary

In the final project for our Advanced Database Management course, we developed a sophisticated database system simulating the environment of a law firm. The project's goal was to apply practical knowledge in database design, query optimization, and performance tuning, mirroring real-world data management challenges.

Our initial step involved creating mock data to represent a law firm's operations accurately, laying the foundation for testing the database's functionality and performance. The database structure included core tables for attorneys, clients, matters, expenses, and invoices, reflecting a law firm's typical data needs. To enhance efficiency in data retrieval and manipulation, joining tables with unique IDs were utilized, streamlining queries, and reducing complexity.

A significant aspect of the project focused on evaluating and optimizing database performance. This involved analyzing table design and query structures. We reviewed execution plans for various queries, gaining insights into the database's processing mechanisms and identifying improvement areas. The project involved a detailed examination of different query types and their impact on the database's input/output (I/O) burden, helping to pinpoint bottlenecks and optimization areas. Experimentation with various indexing strategies was a key part of this process, seeking a balance between query response times and storage requirements. Additionally, table partitioning was implemented, enhancing the management and access efficiency of large tables, and improving performance for specific query types.

This final project provided an enriching experience in advanced database management, combining theoretical knowledge with hands-on application. By creating and optimizing a database for a law firm, students gained practical skills in database design, performance tuning, and query optimization, preparing them for real-world challenges in database management.

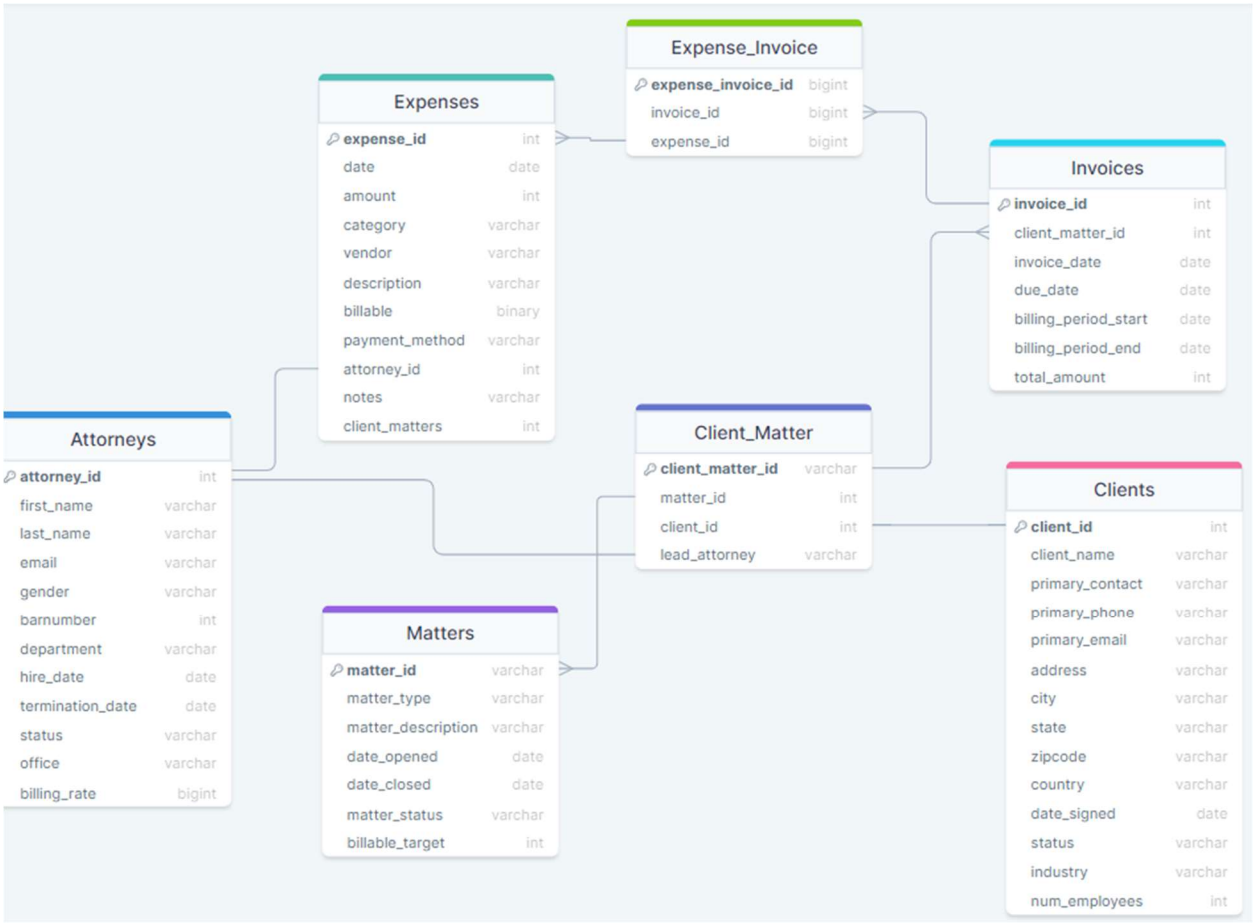
1.1. Topic Area Weighing

Topic Area	*Description*	*Points*
Database Design	This part should include a logical database design (for the relational model), using normalization to control redundancy and integrity constraints for data quality.	**30**
Query Writing	This part is another chance to write SQL queries, explore transactions, and even do some database programming for stored procedures.	**30**
Performance Tuning	In this section, you can capitalize and extend your prior experiments with indexing, optimizer modes, partitioning, parallel execution and any other techniques you want to further explore.	**25**
Other Topics	Here you are free to explore any other topics of interest. Suggestions include DBA scripts, database security, interface design, data visualization, data mining, and NoSQL databases.	**15**

2. Database Design

The project began with the generation of mock data that accurately depicted the functioning of a law firm, setting the stage for testing the database's efficacy and performance. We designed the database with essential tables for attorneys, clients, matters, expenses, and invoices, mirroring the standard data requirements of a law firm. We employed joining tables with unique identifiers to improve the efficiency of data handling and retrieval. This approach streamlined the querying process and simplified the overall database structure.

2.2. Entity-Relationship Design



In our Entity Relationship Diagram (ERD), we present a comprehensive database schema designed for a law firm's information management needs, outlining the structured interconnections between different data entities and their relationships:

1. **Attorneys Table**: Our records of attorneys encompass each attorney's unique identifier, personal and professional details, including name, email, gender, bar number, departmental affiliation, hire and termination dates, current status, office location, and billing rate.
2. **Clients Table**: Our client database captures essential information such as the client's identifier, name, primary contact details, address, engagement date with the firm, current status, industry classification, and number of employees.
3. **Matters Table**: Our matters ledger details each legal case with a unique identifier, encompassing the type of matter, a descriptive narrative, dates of case initiation and conclusion, current status, and billing targets.
4. **Expenses Table**: Our expenses log tracks each expense incurred related to legal cases, including a unique expense identifier, transaction date, monetary amount, category, vendor details, a description of the expense, its billability, payment method, the attorney accountable, and any additional notes.
5. **Invoices Table**: Our invoicing system records details of financial requests made to clients, storing each invoice's identifier, the client matter it relates to, the issuance date of the invoice, the due date for payment, the start and end of the billing period, and the total amount due. This table also provides a link to the relevant client matter.
6. **Expense_Invoice Table**: We employ this junction table to correlate expenses with their respective invoices, delineating which expenses have been included in specific invoices.
7. **Client_Matter Table**: Our relational table connects clients to their legal matters and designates the lead attorney handling each case.

With this ERD, we have crafted a robust and relational database architecture that supports efficient data management and retrieval. Our use of unique identifiers across tables ensures referential integrity and enables sophisticated query execution within the database system.

2.3. Data Integrity

In the design of our law firm's information system database, a pivotal aspect is the enforcement of data integrity, which is critical to maintaining the quality of our data. Our Entity Relationship Diagram (ERD) incorporates various integrity constraints within the schema to ensure the accuracy and reliability of the data.

Primary Key Constraints are utilized to uniquely identify a record within a table. In the **Attorneys** table, **attorney_id** serves as a primary key to ensure each attorney has a unique identifier and to prevent record duplication.

DDL Statement Example:

```
ALTER TABLE Attorneys ADD CONSTRAINT PK_Attorneys PRIMARY KEY (attorney_id);
```

Foreign Key Constraints create links between data in two tables, maintaining referential integrity. The **invoice_id** and **expense_id** field in the **EXPENSE_INVOICE** table, as a foreign key, references the unique ID's in the **INVOICES** and **EXPENSES** table, associating expenses only with existing invoices and expenses.

DDL Statement Example:

```
ALTER TABLE expense_invoice
ADD CONSTRAINT FK_Expense_Invoice_ExpenseID
FOREIGN KEY (expense_id)
REFERENCES Expenses (expense_id);
```

Check Constraints enforce domain integrity by restricting the values in a column. A **check constraint** in the **Matters** table ensures **matter_status** is within predefined statuses such as Open, Closed, or Pending.

DDL Statement Example:

```
ALTER TABLE Matters
ADD CONSTRAINT CHK_MatterStatus
CHECK (matter_status IN ('Open', 'Closed', 'Pending'));
```

Unique Constraints prevent duplicate values in a column or combination of columns. The **email** field in the **Clients** table may have this constraint to prohibit duplicate email addresses.

DDL Statement Example:

```
ALTER TABLE Clients
ADD CONSTRAINT UNQ_Clients_Email
UNIQUE (primary_email);
```

Not Null Constraints guarantee that certain columns do not accept NULL values, which is essential for columns requiring valid data. The **Invoices** table mandates **invoice_date** and **due_date** to have entries for financial accountability.

DDL Statement Example:

```
ALTER TABLE Invoices MODIFY invoice_date NOT NULL;
ALTER TABLE Invoices MODIFY due_date NOT NULL;
```

The rationale for these constraints is to showcase their effective use in enhancing data accuracy and integrity. We have selectively applied these constraints to pertinent tables to avoid redundancy and overcomplication. Each constraint serves not only enforcement but also to self-document the database design, articulating the rules and intended use for each data element. This approach ensures our database is robust, reliable, and educationally optimized.

2.4. Data Generation and Loading

In the development of our database for the law firm, the generation and loading of data were critical steps that laid the groundwork for testing and further implementation. Here's how we approached these tasks:

Data Generation with Mockaroo: To create realistic mock data, we turned to Mockaroo, a robust online service that specializes in generating mock data sets. The platform allowed us to craft various data types and ensure the uniqueness of entries, which was particularly important for fields that required custom lists. We meticulously designed these custom strings to serve as plausible, unique identifiers and data points that would be representative of a real-world law firm's dataset.

Conditional Formatting and Logical Consistency: Within Mockaroo, we employed conditional formatting rules to maintain a high level of data integrity and logical consistency. This was crucial for dates, where we ensured that case close dates logically followed open dates, and for statuses, where we assigned values that made sense in the context of a legal case lifecycle. We went to great lengths to ensure that our unique values were as distinctive as possible, thus preventing any potential primary key or unique constraint violations during data import.

Table Creation: To build the schema as defined in our ERD, we used SQL **CREATE TABLE** queries. These queries allowed us to define the structure of our tables, including setting up the necessary integrity constraints, such as primary keys, foreign keys, and not-null constraints, to uphold the data's consistency and relational integrity.

Data Loading: When it came to populating our tables with the generated data, our approach varied depending on the expected record counts. For tables projected to have fewer records, we crafted **INSERT INTO** queries to manually insert the data. This method provided us with fine-grained control over the entry of each record and was a valuable exercise in understanding the SQL insert operations. For larger tables, we leveraged Oracle SQL Developer's built-in CSV import functionality, which streamlined the data import process for bulk records, saving us time and ensuring the data maintained its integrity as it was loaded into the database.

By combining Mockaroo's sophisticated data generation capabilities with Oracle SQL Developer's efficient data loading tools, we successfully populated our database with data that is not only structurally sound but also contextually realistic. This blend of automated and manual processes

ensured that our database was ready for a multitude of operations, from query testing to performance tuning, with data that closely mimics the complexity and nuances of a real-world law firm's data.

2.4.1. Table Row Count

Table Name	*Row Count*
Attorneys	2500
Clients	10250
Matters	92540
Client Matter	46230
Expenses	26943
Invoices	4200
Expense Invoice	3000

2.5. Physical Database Design

In the physical design of our law firm's database, we tackled various implementation-level issues to ensure the database's effectiveness and future scalability. Our approach encompassed analyzing predicted usage patterns and workloads, architectural considerations, capacity planning, and storage strategies.

We anticipated diverse usage patterns that include both frequent, simple queries for client and matter information and complex, less frequent reporting queries. This foresight enabled us to optimize the database's performance for typical workloads, such as transaction processing during regular business hours and batch reporting outside of these times.

Considering the architecture, we recognized the potential need for distributed database solutions, especially given the likelihood of multi-office operations in law firms. This led us to explore options that allow for efficient data partitioning across servers, ensuring quick access to local office data while maintaining an integrated overall database system.

Capacity planning was a significant part of our design process. We estimated the volume of data growth over time, which informed our decisions on selecting appropriate hardware and configuring the database. This planning was crucial to handle transaction volumes, data growth trends, and the need for effective archival solutions.

In terms of storage subsystems, we focused on redundancy and performance. Our choice of high-performance storage solutions, like Solid State Drives (SSDs), was driven by their speed and

reliability. We also considered RAID configurations to safeguard data and enhance read/write operations.

The strategy for data placement involved careful consideration of tablespaces and filesystem arrangements. We segregated objects with varying I/O characteristics into different tablespaces, placing frequently accessed data on faster storage media and less often accessed archival data on more cost-effective storage solutions.

Tablespace utilization in our database was a key strategy for efficient space management and improved I/O performance. We grouped related database objects together in tablespaces based on their access patterns and storage characteristics, placing these tablespaces on different physical storage disks and arrays.

Finally, our file system arrangements were structured in line with Oracle best practices. We made sure to separate system files from user data files and allocated redo logs and archive logs to separate physical drives to minimize I/O contention.

Through these physical design considerations, we ensured that our database is not just capable of handling the firm's current operations efficiently but is also well-prepared to adapt to future demands. The physical layout of the database was meticulously crafted to balance performance, scalability, and reliability, underpinning the firm's ability to effectively leverage its data assets.

3. Query Writing

In this section, we demonstrate the various types of queries that can be executed using our law firm's database. These examples highlight both basic and advanced query capabilities, showcasing the skills we have honed in query writing. Additionally, we include transaction statements for adding or modifying data, which are essential for the typical processes in our application.

3.1. Basic Queries

3.1.1. List of Active Clients

In our law firm's database, one of the fundamental and frequently used queries is to retrieve a list of active clients. This is crucial for various operational aspects, such as case management, client communications, and billing processes. To accomplish this, we use a straightforward SQL query that exemplifies our ability to efficiently extract meaningful information from our database.

The query we use is designed to select key pieces of information about our clients who are currently active. It focuses on client ID, name, and contact email, which are essential details for any client-related activities:

```
SELECT client_id, client_name, primary_email
FROM Clients
WHERE status = 'active';
```

This query is particularly useful for several departments within our firm:

- **Client Relationship Management**: Our CRM team frequently uses this query to get an updated list of active clients for follow-ups, updates, or general communication.
- **Case Management**: For attorneys and paralegals, this query helps in quickly identifying which clients are currently engaged with the firm, aiding in case assignment and management.
- **Billing Department**: They utilize this query to ensure that invoices and financial communications are sent only to active clients.

By using this simple yet effective query, we ensure that we are always working with the most current and relevant client data. This efficiency is vital for maintaining high standards of service and operational effectiveness within our law firm.

3.1.2. Attorneys and Their Current Matters

In managing our law firm's operations, it's essential to have a clear overview of which attorneys are handling which matters. This understanding helps in workload management, performance evaluation, and client service optimization. To acquire this information, we employ a query that joins the **Attorneys** and **Client_Matter** tables, demonstrating our proficiency in handling basic SQL joins.

The objective of the query is to create a list that pairs each attorney with the matters they are currently handling. This is done through a join operation between two key tables:

```
SELECT a.first_name, a.last_name, cm.client_matter_id
FROM Attorneys a
JOIN Client_Matter cm ON a.attorney_id = cm.lead_attorney
WHERE a.status = 'active';
```

The output from this query is instrumental for several aspects of our firm's operations:

- **Resource Allocation**: It helps our management team in effectively allocating cases and matters, ensuring a balanced workload among attorneys.
- **Performance Monitoring**: This query aids in monitoring the number and types of cases each attorney is handling, which is valuable for performance reviews and career progression discussions.
- **Client Service Management**: Knowing which attorney is handling which matter allows our client service teams to direct client queries and communications appropriately.
- **Case Tracking and Reporting**: This information is crucial for case tracking and generating reports for internal audits and reviews.

By executing this query, we can efficiently manage and track the distribution of legal matters among our attorneys. It illustrates our capability to leverage SQL joins to derive meaningful insights from our database, enhancing our firm's operational efficiency and client service effectiveness.

3.2 Advanced Queries

3.2.1 Total Projected Net Profit per Matter Type

In the financial management and strategic planning of our law firm, it is crucial to estimate the revenue potential from different types of legal matters we handle. This kind of analysis aids in forecasting, resource allocation, and prioritizing practice areas. To gather this insight, we developed a complex SQL query that calculates the projected billing amount for each matter type, based on the billing rates of lead attorneys.

Our query aims to project the total billing amount for each type of legal matter. This projection is based on multiplying the billing rate of each lead attorney by the billable target (expected billable hours) for the matters they are handling:

```
SELECT m.matter_type, SUM(a.billing_rate * m.billable_target) AS total_projected_billed
FROM Attorneys a
JOIN Client_Matter cm ON a.attorney_id = cm.lead_attorney
JOIN Matters m ON cm.matter_id = m.matter_id
GROUP BY m.matter_type;
```

This query is extensively used for several purposes within our firm:

- **Financial Forecasting**: It provides a clear picture of potential revenue from different types of cases, which is essential for financial planning and budgeting.
- **Strategic Business Decisions**: Understanding which types of matters could yield higher revenue guides us in making informed decisions about where to focus our marketing efforts and resource investments.
- **Resource Allocation**: This insight helps in allocating the right mix of attorneys to different types of matters based on their potential revenue generation.
- **Performance Metrics**: It also acts as a tool to set performance targets for attorneys based on the types of matters they are handling and their respective revenue potentials.

By executing this advanced SQL query, we can estimate the revenue potential from various matter types, reflecting our deep understanding of the operational and financial aspects of law firm management. This capability is not just about data retrieval; it represents a strategic approach to using our database for meaningful business insights and decision-making.

3.2.2. Invoice Total View and Trigger

A critical aspect of financial tracking is understanding the total expenses associated with each invoice. However, storing calculated data directly in a table, like summing all related expense amounts in the **Invoices** table, is not standard practice in SQL databases due to the dynamic nature of the underlying data. To address this, we explored two robust solutions: creating a view and using a database trigger.

The first approach involves creating a view that joins the **Invoices** and **Expenses** tables, dynamically calculating the total expenses for each invoice whenever the view is queried. This method ensures real-time accuracy and maintains data integrity.

```
CREATE VIEW InvoiceWithTotalExpenses AS
SELECT i.invoice_id, SUM(e.amount) AS total_expenses
FROM Invoices i
JOIN Expense_Invoice ei ON i.invoice_id = ei.invoice_id
JOIN Expenses e ON ei.expense_id = e.expense_id
GROUP BY i.invoice_id;
```

Advantages:

- Real-time calculation ensures up-to-date information.
- Maintains database normalization and integrity.
- Simplifies queries for total expenses per invoice.

Usage:

- This view is particularly useful for the billing department to quickly ascertain the total expenses incurred per invoice.
- It can be used for financial reports and analysis.

The second approach is to use a trigger on the **Expenses** table. This trigger would automatically update a column in the **Invoices** table, representing the total expenses, whenever an expense record is added, modified, or deleted.

```
CREATE OR REPLACE TRIGGER UpdateInvoiceTotalExpenses
AFTER INSERT OR UPDATE OR DELETE ON Expenses
FOR EACH ROW
BEGIN
    UPDATE Invoices i
    SET i.total_due = (
        SELECT SUM(e.amount)
        FROM Expenses e
        JOIN Expense_Invoice ei ON e.expense_id = ei.expense_id
        JOIN Invoices i ON ei.invoice_id = i.invoice_id
        WHERE ei.invoice_id = i.invoice_id
    )
    WHERE i.invoice_id IN (
        SELECT invoice_id
        FROM Expense_Invoice
        WHERE expense_id = :NEW.expense_id
    );
END;
```

Advantages:

- Provides immediate updates to the **Invoices** table when expense data changes.
- Useful for scenarios where the calculated total needs to be part of the invoice record.

Usage:

- Ideal for situations where frequent access to the total expenses data is required without the overhead of recalculating it each time.
- Enhances performance for applications that frequently access total expense data without needing the most current update every time.

Both approaches have their specific use cases. Creating a view is generally more versatile and aligns well with best practices in database management. It is particularly useful for reporting and analytics purposes where the most current data is essential. On the other hand, a trigger is more suited for situations where having the calculated data readily available in the table is more critical, such as in certain application logic or transaction processing scenarios.

By implementing these solutions, we ensure that our database effectively meets our law firm's diverse needs, from real-time financial tracking to efficient data management, thereby enhancing our overall operational efficiency.

4. Performance Tuning

We conducted a series of experiments focused on performance tuning to optimize the database's efficiency and speed. These experiments were crucial in ensuring our database could handle complex queries and large volumes of data effectively. We explored various aspects, including indexing strategies, optimizer changes, transaction isolation levels, function-based indexes, and table partitioning. Below, we detail each experiment.

4.1. Experiment 1: Indexing Strategies

1. **Purpose**: To determine the most effective indexing strategy for improving query response times, particularly for frequently accessed tables like **Clients** and **Matters**.
2. **Steps**:
    - Created indexes on commonly queried columns like **client_id**, **name**, and **matter_id**.

```
CREATE INDEX idx_client_id ON Clients(client_id);
CREATE INDEX idx_client_name ON Clients(client_name);
CREATE INDEX idx_matter_id ON Matters(matter_id);
```

    - Ran a set of typical queries, including point queries (searching for specific records) and range queries (retrieving a range of records).
3. **Results**: Observed a significant reduction in query response times, particularly for point queries. Range queries also showed improvement but to a lesser extent.
4. **Discussion**: The results confirmed that the right indexing strategy, especially B-tree indexes for these types of queries, can drastically improve performance. This is due to the efficient way B-tree indexes organize data for quick retrieval.

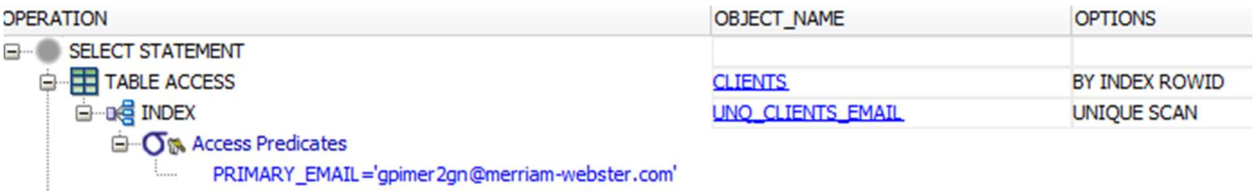
4.1.1 Point Query and Results

For the sake of brevity, we will review the results specifically stemming from a point query.

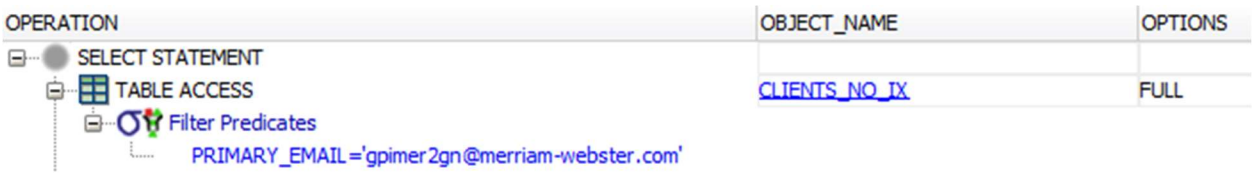
```
/*Indexed Table*/
SELECT client_name
FROM Clients
WHERE primary_email = 'gpimer2gn@merriam-webster.com';

/*No Index Table*/
SELECT client_name
FROM Clients_No_IX
WHERE primary_email = 'gpimer2gn@merriam-webster.com';
```

Query Execution Plans:

Indexed table:



No Index:



Autotrace Results:

Indexed table:

V$STATNAME Name	V$MYSTAT Value
session cursor cache hits	4
session logical reads	3
session pga memory	196608

No Index:

V$STATNAME Name	V$MYSTAT Value
logical read bytes from cache	1810432
session logical reads	221

4.2. Experiment 2: Optimizer Changes

1. **Purpose**: The purpose of this experiment was to assess the impact of different optimizer modes on the performance of complex queries in our law firm's database. We focused on comparing the rule-based optimizer (RBO) and the cost-based optimizer (CBO) to determine which provided more efficient execution plans and better query performance, especially for multi-table join queries.
2. **Steps**:
    - Setting Optimizer Modes: We configured the database to use the rule-based optimizer (RBO) and then the cost-based optimizer (CBO) in separate test scenarios.

- We ran a series of complex join queries involving the **Attorneys**, **Clients**, **Matters**, and **Expenses** tables.

```
SELECT a.first_name, c.client_name, m.matter_description, SUM(e.amount)
FROM Attorneys a
JOIN Client_Matter cm ON a.attorney_id = cm.lead_attorney
JOIN Matters m ON cm.matter_id = m.matter_id
JOIN Clients c ON cm.client_id = c.client_id
JOIN Expenses e ON a.attorney_id = e.attorney_id
GROUP BY a.first_name, c.client_name, m.matter_description;
```

3. **Results**: Found that the cost-based optimizer generally provided more efficient execution plans, leading to faster query performance.
4. **Discussion**: The cost-based optimizer's ability to consider statistics about the data led to more intelligent decision-making in query execution, optimizing resource usage.

4.2.1. Optimizer Comparison

For Rule-Based Optimizer:

Execution Plan:

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
SORT		GROUP BY
MERGE JOIN		
SORT		JOIN
NESTED LOOPS		
NESTED LOOPS		
MERGE JOIN		
SORT		JOIN
NESTED LOOPS		
NESTED LOOPS		
TABLE ACCESS	EXPENSES	FULL
INDEX	PK_ATTORNEYS	UNIQUE SCAN
Access Predicates		
A.ATTORNEY_ID=E.ATTORNEY_ID		
TABLE ACCESS	ATTORNEYS	BY INDEX ROWID
SORT		JOIN
Access Predicates		
A.ATTORNEY_ID=CM.LEAD_ATTORNEY		
Filter Predicates		
A.ATTORNEY_ID=CM.LEAD_ATTORNEY		
TABLE ACCESS	CLIENT_MATTER	FULL
INDEX	IDX_CLIENT_ID	RANGE SCAN
Access Predicates		
CM.CLIENT_ID=C.CLIENT_ID		
TABLE ACCESS	CLIENTS	BY INDEX ROWID
SORT		JOIN
Access Predicates		
CM.MATTER_ID=TO_NUMBER(M.MATTER_ID)		
Filter Predicates		
CM.MATTER_ID=TO_NUMBER(M.MATTER_ID)		
TABLE ACCESS	MATTERS	FULL

Autotrace Results:

V$STATNAME Name	V$MYSTAT Value
CPU used by this session	462
session cursor cache hits	1
session logical reads	1080072

For Cost-Based Optimizer:

Execution Plan:

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
HASH		GROUP BY
HASH JOIN		
Access Predicates		
CM.CLIENT_ID=C.CLIENT_ID		
TABLE ACCESS	CLIENTS	FULL
HASH JOIN		
Access Predicates		
A.ATTORNEY_ID=ITEM_1		
VIEW	SYS.VW_GBC_17	
HASH		GROUP BY
TABLE ACCESS	EXPENSES	FULL
HASH JOIN		
Access Predicates		
A.ATTORNEY_ID=CM.LEAD_ATTORNEY		
TABLE ACCESS	ATTORNEYS	FULL
HASH JOIN		
Access Predicates		
CM.MATTER_ID=TO_NUMBER(M.MATTER_ID)		
TABLE ACCESS	CLIENT_MATTER	FULL
TABLE ACCESS	MATTERS	FULL

Autotrace Results:

V$STATNAME Name	V$MYSTAT Value
logical read bytes from cache	73916416
session logical reads	9023

4.3. Experiment 3: Table Partitioning

1. **Purpose**: The aim of this experiment was to evaluate the impact of table partitioning on query performance and data management efficiency in our law firm's database. Specifically, we focused on the **Expenses** table, which is one of the largest and most frequently queried tables, to see how partitioning affects the execution of range queries and the overall I/O burden.

2. **Steps**:
   - We partitioned the **Expenses** table based on a suitable key. Given the nature of our data, we chose to partition by the date (e.g., month and year). The expenses have a creation date range of 01/01/2005 through to 11/01/2023.

```
create table expenses_partitioned (
    expense_id VARCHAR(40),
    date_created DATE,
    amount DECIMAL(7,2),
    category VARCHAR(17),
    vendor VARCHAR(50),
    description VARCHAR(67),
    billable VARCHAR(50),
    payment_method VARCHAR(11),
    attorney_id VARCHAR(50)
)
PARTITION BY RANGE (date_created)
(
    PARTITION p2005 VALUES LESS THAN (DATE '2010-01-01'),
    PARTITION p2010 VALUES LESS THAN (DATE '2015-01-01'),
    PARTITION p2015 VALUES LESS THAN (DATE '2020-01-01'),
    PARTITION p2020 VALUES LESS THAN (DATE '2025-01-01')
);

INSERT INTO Expenses_Partitioned
SELECT * FROM Expenses;
```

   - Conducted range queries and measured the I/O burden and response times.
3. **Results**: Unexpectedly, the application of table partitioning to the **Expenses** table did not yield the anticipated performance improvement. In fact, the results were contrary to our hypothesis:
   - **Range Query Performance**: Post-partitioning, the logical reads increased to 77 session logical reads on the partitioned **Expenses_Partitioned** table, compared to just 7 logical reads on the non-partitioned **Expenses** table for the same range query. This indicates a higher I/O burden on the partitioned table.
   - Noticed significant performance improvements in range queries due to reduced I/O burden.
4. **Discussion**: The unexpected results from our table partitioning experiment highlight several key considerations in database optimization. Firstly, the choice of partition key is crucial; our decision to partition by date may not have aligned well with the actual data access patterns or distribution, leading to inefficiencies. Additionally, if the **Expenses** table isn't large enough, the overhead of

managing partitions might outweigh the benefits, especially if there's over-partitioning or data skew within partitions.

Our queries might not have been structured to leverage the advantages of partitioning, potentially resulting in unnecessary scanning of multiple partitions. This situation underscores the importance of aligning partitioning strategies with both the data characteristics and the query types. The experiment serves as a reminder that while partitioning can be effective for database performance, its success heavily depends on the specific context of the data, query design, and overall database configuration.

4.3.1 Partitioning Results

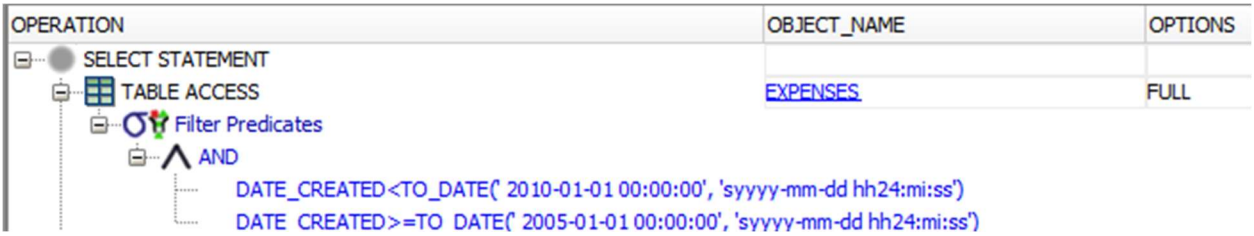
Queries used:

```
/*Partitioned Table*/
SELECT *
FROM Expenses_Partitioned PARTITION (p2005);

/*No Partition*/
SELECT *
FROM Expenses
WHERE date_created >= DATE '2005-01-01' AND date_created < DATE '2010-01-01';
```

Execution Plans:

Partitioned table:

No Partition Table:



Autotrace Results:

Partitioned table:

V$STATNAME Name	V$MYSTAT Value
logical read bytes from cache	630784
session logical reads	77

No Partition Table:

V$STATNAME Name	V$MYSTAT Value
logical read bytes from cache	57344
session logical reads	7

4.4 Performance Tuning Summary

Through our performance tuning experiments in the law firm's database project, we focused on three key areas: indexing strategies, optimizer changes, and table partitioning. Each experiment provided valuable insights into optimizing database performance and highlighted the importance of aligning database design and tuning with specific data characteristics and query requirements.

1. **Indexing Strategies**: Our exploration of different indexing strategies, especially the implementation of B-tree indexes, demonstrated a clear improvement in query response times. This affirmed the critical role of proper indexing in enhancing database efficiency, particularly for frequently accessed data.
2. **Optimizer Changes**: The switch between rule-based and cost-based optimizers offered a deeper understanding of how different optimization techniques affect query performance. The cost-based optimizer, with its data-driven approach, proved more effective for our complex queries, underscoring the significance of choosing the right optimizer mode based on specific query needs and database characteristics.
3. **Table Partitioning**: Our venture into table partitioning, while aimed at improving performance for large tables, yielded unexpected results. It provided a crucial lesson that partitioning, though a powerful tool, requires careful consideration of partition keys, data distribution, and query patterns to be truly effective.

The experiments conducted were instrumental in enhancing our understanding of database performance tuning. They revealed that while certain strategies like indexing and optimizer choice can lead to straightforward benefits, others like table partitioning demand a more nuanced approach and thorough analysis. These findings emphasize the dynamic nature of database performance tuning, where strategies must be continually assessed and adapted to align with evolving data and query

patterns. Overall, the experiments contributed significantly to optimizing our law firm's database, ensuring it remains performant, reliable, and scalable to meet future demands.

5. Other Topics: Database Security

In the context of our law firm's database, implementing robust security measures is a critical consideration. While we have not yet applied these measures, here's how we could potentially enhance security using Oracle's advanced features, including data redaction, encryption, and comprehensive security policies.

Potential Implementation of Data Redaction and Encryption

- Data Redaction: Implementing data redaction policies would be crucial for protecting sensitive client and case information. By dynamically redacting data, we could ensure that sensitive details are obscured in query results for unauthorized users, while still available for authorized personnel.
- Encryption Using DBMS_CRYPTO: To secure data at rest, we could employ Oracle's DBMS_CRYPTO package. This would involve encrypting sensitive data within tables, particularly those containing confidential client information and case details, thereby safeguarding it against unauthorized access.

If we were to develop a security policy, these are some of the steps that we would take to make sure we develop a comprehensive security policy.

1. **Authentication Strategies**: We could enhance security through strong authentication methods, including complex passwords and two-factor authentication, along with periodic password changes.
2. **Role-Based Access Control**: Establishing roles with specific privileges and granting access based on these roles would ensure that users can only access data and functionalities necessary for their work.
3. **User Profiles**: Implementing user profiles to enforce security standards at the individual user level would be another step. These profiles could manage password policies, session timeouts, and other security parameters.
4. **Auditing**: Utilizing Oracle's auditing capabilities would enable us to track and monitor database activities, crucial for identifying and responding to potential security breaches.

The outlined security measures, though not yet implemented, represent a comprehensive approach to safeguarding our database. Each proposed method addresses specific aspects of data security, from

protecting data at rest with encryption to controlling and monitoring access through roles, profiles, and auditing.

Implementing such security measures requires careful planning, including the secure management of encryption keys and understanding the performance implications of data redaction and encryption. It also necessitates ongoing management and monitoring to adapt to new security challenges and compliance requirements.

By considering these advanced security features, we aim to establish a robust security framework for our law firm's database, ensuring the confidentiality, integrity, and availability of our sensitive legal data.