Student Finance Database

Charan Kumar Nara
cnara
50545001
Email: cnara@buffalo.edu

Adarsh Reddy Bandaru
adarshre
5057208
Email: adarshre@buffalo.edu

Dharma Acha
dharmaac
50511275
Email: dharmaac@buffalo.edu

Abstract**—** In the field of higher education, understanding and managing student finances effectively are paramount, which stands at higher priority for ensuring academic success and financial well-being. This report proposes the design and implementation of a robust database system titled "Student Finance Database" which mainly focuses on centralizing and organizing student finance data to facilitate and manage the data. The key objective of the proposed database system includes efficient data collection, secure storage. Through a relational database schema designed for scalability and performance, the system supports query interfaces and reporting tools for analyzing student finance data based on diverse criteria.

1. Introduction

In the perspective of higher education, the management of student finances stands in pivotal position which influences academic achievement, student retention and overall well-being. As the cost of pursuing higher education continues to increase globally, understanding the dynamics of the student spending habits, financial needs, and patterns becomes very important. However, effectively capturing, organizing and analysing large volumes of student finance data rises significant challenges, which necessitating innovative approaches to database management systems.

The dataset contains data representing the spending habits of 10000 students across different demographic groups and academic backgrounds. And the dataset contains the information about age, gender, year in school, major, monthly income, financial aid received, and expenses in different spending categories.

The spending categories also consist of tuition, housing, food, transportation, books and supplies, entertainment, personal care, technology, health and wellness and miscellaneous expenses. The dataset also consists of preferred payment methods for each student.

The main objectives of the database system include efficient data collection, secure storage and optimized retrieval mechanisms to enable users which includes educational institutions, financial advisors to deliver actionable insights. This system will facilitate efficient data retrieval, analysis and reporting.

2. Problem Statement

The project mainly focuses on "Student Finance database". Its main purpose is to design and implement a robust and efficient database system to manage and analyze student expenses based on different demographic groups and academic backgrounds.

3. Background of the Problem

The problem of managing student finances in higher education is increasing attention in recent years. Several factors contribute to the complexity of this challenge, including the rising cost of tuition, the growing student debt, and the diverse financial backgrounds of students. Understanding the background of this problem talks about the necessity of developing a robust database system which is used to student finance management.

The approach of managing databases often involves in disparate systems and process that are not well integrated or standardized, this leads to inefficiencies and limitations in data management.

Here are some challenges that affect efficient data collection, analysis and reporting.

a. Lack of Comprehensive Data Management Systems

When it comes to handling student's finance in educational institutions, the old-fashioned methods are not very effective. Basically, the department keeps its own records using paper or different computer systems. So, this makes it hard to keep track of all the information about how students are spending money like on tuition, housing or books. Because the information is spread out. It's tough to collect, analyze and understand.

b. Lack of Centralized Data Storage

Student finance data often stored in separate databases or spreadsheets maintained by different departments like financial aid office, student accounts and academic departments. Without the centralized repository for storing and organizing this data, there is risk of duplication of data.

c. Inconsistent Data Quality and Accuracy

When data is stored in disparate systems and formats, ensuring the quality and accuracy of student finance becomes data becomes challenging. Data entry errors, inconsistency in coding schemas and outdated data information can compromise the reliability of the analysis and reporting, this led to false conclusions and ineffective decision-making.

4. OBJECTIVE

The project deals in understanding the challenges by developing a centralized and structured Student Finance system database. By building a database system we aim to achieve:

a. Efficient Data Management

We will create a robust database platform for storing and organizing student finance data, which reduces the risk of data duplications and further improving the data consistency.

b. Data Analysis

With the help of centralizing data, this database system allows more sophisticated and efficient data analysis, providing valuable insights into student finance performance,

c. Scalability and Performance

Design the database system to scale efficiently to accommodate growing datasets and user traffic. Implementing performance optimization techniques like query optimization, caching and database partitioning, to ensure responsive and scalable system performance

d. Maintenance and Support

Establishing procedures for databases maintenance, backup and recovery to ensure system reliability and data integrity. We provide ongoing support and updates to address any issues, this enhances system functionality and incorporate the feedback from the users.

Target Users

The implemented database system is used by many categories of people. Here are some real-life examples.

a. Student services departments

Student services departments utilize this database systems to provide support services which is related to financial literacy, budgeting and financial counseling.

b. Academic advisors

They use this database systems to understand the financial constraints and challenges faced by students and provide guidance on academic planning and course selection. They may also use this database to identify resources and opportunities for financial assistance.

c. Educational Institutions

Educational institutions, including universities, colleges and schools are primary users of the student finance databases. They utilize this database to track and manage student finances, analyzing spending patterns.

d. Institutional Researches

Institutional researches within educational institutions make use of this database systems to conduct analyses and generate report on student finance trends, graduation rates and other performance indicators.

5. FUTURE SCOPE

By taking these implementations in future directions, the student financial database systems can play pivotal role in promoting financial literacy, equity and student success in higher education. Here are some of the future trends

a. Personalized Financial guidance

The implementations of the personalized financial guidance tools within the database systems can empower students to make informed financial decisions and individual circumstances.

b. Predictive Analytics

The future iterations of student finance database systems can incorporate predictive analytics techniques to show student financial needs, identify at-risk students. This data is used in machine learning algorithms and predictive models which predicts financial needs of the students and helps in managing finances and improving academic outcomes.

c. Mobile and Cloud-Based Solutions

Future student finance database systems may harness mobile and cloud-based technologies to improve accessibility and user experience for students and administrators. Mobile apps and web apps would offer convenient access to financial data, interactive tools and educational materials which increases engagement among students.

6. Decomposing the original table to comply with the Boyce - Codd Normal Form (BCNF)

Dataset: datasetLink

We have taken the dataset from kaggle which contained one table. To cater to the requirements of the projects we will be scaling, and creating a dummy table which will be an extension to the original dataset.

We initially had two tables, the Student_information table with 20 attributes and the StudentFinancialRecords table with 14 attributes. Both of them have multiple BCNF violations which we will handle using decomposition. The monthly expense attributes such as Tuition, Housing, Food, etc., partially depend on the StudentID along with the Month and Year. However, these expenses are not dependent on other student attributes such as Age, Gender, etc. This implies that if we considered (StudentID, Month, Year) as a composite key for the entire record, then Monthly Income and Financial Aid would partially depend on this composite key, which violates 2NF. If we were to derive from fields within the expenses like the total_expense column that sums all the individual expenses, these would depend on other non-key attributes Tuition, Housing, Food, etc., which in turn depend on the composite key (StudentID, Month, Year), resulting in transitive dependencies and violating 3NF.

Similarly, the StudentFinancialRecords table also has anomalies and violations. If a student gets multiple forms of financial aid, their basic information (StudentID, AcademicYear) needs to be repeated for each record. This redundancy can lead to inconsistent data if updates are not properly managed. The table includes attributes that are only partially dependent on the primary key. For example, ProgramName, Department, and HourlyRate are related to work-study programs and do not directly relate to other financial aid types like scholarships or loans.

To tackle these dependencies, we decomposed the original tables into 12 tables to comply with BCNF, 3NF and 2NF rules.

6.1. Students Table

Attributes: StudentID (PK), Age, Gender, YearInSchool, Major

Functional Dependencies: StudentID → Age, Gender, YearInSchool, Major

BCNF: The table is in BCNF because StudentID is a candidate key, and all other attributes are fully functionally dependent on it. There are no partial or transitive dependencies.

6.2. Monthly_income Table

Attributes: MonthlyIncomeID(PK), StudentID (FK), MonthlyIncome, Month, Year

Functional Dependencies: MonthlyIncomeID → StudentID, MonthlyIncome, Month, Year

BCNF: The table is in BCNF as MonthlyIncomeID serves as a superkey, meaning all attributes are fully functionally dependent on it. There are no partial or transitive dependencies.

6.3. Category_mapping Table

Attributes: CategoryID (PK), CategoryName

Functional Dependencies: CategoryID → CategoryName

BCNF: The table is in BCNF because CategoryID is a candidate key. Each category name is uniquely determined by its CategoryID, and there are no partial or transitive dependencies.

6.4. Student_spending Table

Attributes: SpendingID (PK), StudentID (FK), CategoryID (FK), Month, Year, Amount

Functional Dependencies:

SpendingID → StudentID, CategoryID, Month, Year, Amount

BCNF: This table is in BCNF because SpendingID is a superkey, and all attributes are fully functionally dependent on it. The combination of StudentID, CategoryID, Month, and Year forms a composite key that ensures full functional dependency, eliminating partial and transitive dependencies.

6.5. Payment_methods Table

Attributes: MethodID (PK), MethodName

Functional Dependencies: MethodID → MethodName

BCNF: This table is in BCNF as MethodID is the candidate key. The method name is uniquely determined by the method ID, and there are no other dependencies.

6.6. Payment_preferences Table

Attributes: StudentID (FK), MethodID (FK)

Functional Dependencies: None (as there are no non-key attributes)

BCNF: The table is in BCNF. The composite key (StudentID, MethodID) means that each entry in the table is unique and there are no partial or transitive dependencies within this table.

6.7. Academic-years Table

Attributes: AcademicYearID (PK), AcademicYear

Functional Dependencies: AcademicYearID → AcademicYear

BCNF: The table is in BCNF because AcademicYearID is a candidate key, and AcademicYear is fully functionally dependent on it. There are no partial or transitive dependencies.

6.8.. Scholarships Table

Attributes: ScholarshipID (PK), ScholarshipName, ScholarshipAmount

Functional Dependencies: ScholarshipID → ScholarshipName, ScholarshipAmount

BCNF: The table is in BCNF because ScholarshipID is a candidate key, and both Scholarship Name and ScholarshipAmount are fully functionally dependent on it.

6.9. Grants Table

Attributes: GrantID (PK), GrantName, GrantAmount

Functional Dependencies: GrantID → GrantName, GrantAmount

BCNF: The table is in BCNF because GrantID is a candidate key, and both GrantName and GrantAmount are fully functionally dependent on it.

6.10. Loans Table

Attributes: LoanID (PK), LoanProvider, LoanAmount, InterestRate

Functional Dependencies: LoanID → LoanProvider, LoanAmount, InterestRate

BCNF: The table is in BCNF because LoanID is a candidate key, and LoanProvider, LoanAmount, and InterestRate are all fully functionally dependent on it.

6.11. Workstudy_Programs Table

Attributes: WorkStudyID (PK), ProgramName, Department, HourlyRate, HoursPerWeek

Functional Dependencies: WorkStudyID → ProgramName, Department, HourlyRate, HoursPerWeek

BCNF: The table is in BCNF because WorkStudyID is a candidate key, and ProgramName, Department, HourlyRate, and HoursPerWeek are fully functionally dependent on it.

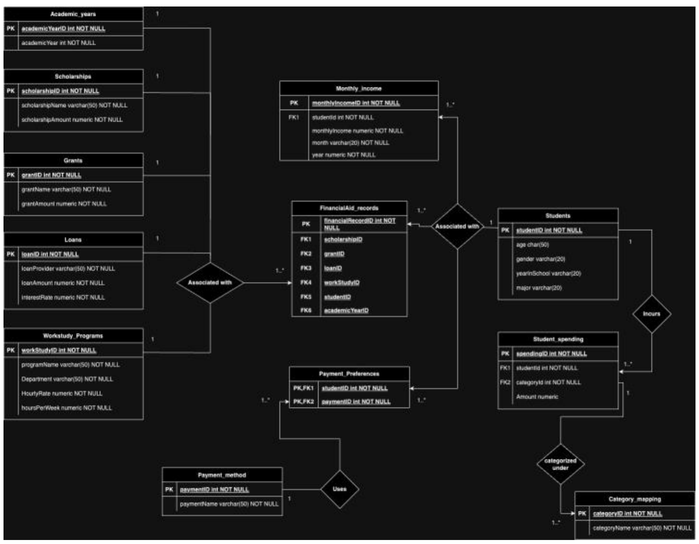
6.12. FinancialAid_Records Table

Attributes: FinancialRecordID (PK), StudentID (FK), AcademicYearID (FK), ScholarshipID (FK, nullable), GrantID (FK, nullable), LoanID (FK, nullable), WorkStudyID (FK, nullable)

Functional Dependencies: FinancialRecordID → StudentID, AcademicYearID, ScholarshipID, GrantID, LoanID, WorkStudyID
BCNF: This table is in BCNF because FinancialRecordID is a superkey. It uniquely identifies each financial record. The use of nullable foreign keys allows for the absence of a particular type of aid.

This split will help in handling repeating groups by segregation of data into different tables. By creating and utilizing id attributes such as MethodID, SpendingID, and FinancialID, we removed partial and transitive dependencies.

7. Entity Relationship diagram



7.1. Entities

There are total twelve entities designed to develop the SQL database for this problem. The entities and their dependencies are discussed under the section 6 on Normalization.

7.2. Cardinality and Relationship:

From the above ER diagram, we can identify some of the cardinalities and relationships between the entities.

Relationships:

- Uses, captures the relationship between Payment_method and Payment_preferences, where each payment preference uses a specific payment method.
- Categorized under, captures the relationship between Student_spending and Category_mapping entities, where each spending record falls under specific category.
- Associated with, captures the relationship between FinancialAid_records and Students, where financial records are associated with a specific student.
- Incurs, captures the relationship between the student_spending and students table, where a student is subject to various spending.

Cardinalities:

For instance, students entity has one to many mapping with Monthly_income, FinancialAid_records, Student_spending and Payment_preferences. In addition to this the Financial_records relation has many to one mapping with Academic_years, Scholarships, Grants, Loans and Workstudy_programs relations.

Task-5

Handling large datasets in the database applications often introduces several challenges, such as performance bottlenecks and increased response time for queries. Here are some challenges we faced

Problems Encountered with Large Datasets

Query Performance: As the size of your data grows, simple queries can become slower if they have to scan large volumes of data.

Indexing Overhead: While indexes are crucial for speeding up query performance, they also add overhead during data insertion and updates as each entry needs to be indexed.

Maintenance Overhead: Large databases require more intensive maintenance, including backups, index rebuilding, and vacuuming in some database systems like PostgreSQL.

Resource Utilization: Larger datasets consume more memory, CPU, and storage, which can lead to resource contention among different processes.

Solutions and Implementations

Optimized Indexing: Selective Indexing: Implement indexes on columns that are frequently used in WHERE clauses, JOIN conditions, or as part of an ORDER BY.

Multi-column Indexes: For queries involving multiple columns, multi-column indexes can be more effective than single-column indexes.

Partial Indexes: These indexes are created only for a subset of a table, typically used when querying only a small portion of the table regularly.

B-Tree Index: In PostgreSQL, the B-Tree index is the standard and most frequently used type of index. It is particularly effective for data with high cardinality, such as columns that contain unique or nearly unique values.

Vertical Partitioning: Splitting a table by columns — for example, storing frequently accessed columns separately from infrequently accessed ones to reduce I/O.

Query Optimization: Analyzing Query Plans: Use EXPLAIN plans to understand how queries are executed and optimize them by rewriting or adjusting joins and selections.

Materialized Views: These are useful for caching the result of a query physically; this can significantly speed up queries that compute aggregates or join several tables.

Data Caching: Implement application-level caching mechanisms such as Redis or Memcached to store frequently accessed data, reducing the number of times the database needs to be queried.

Asynchronous Processing: For operations that do not require immediate consistency (like logging or statistical computations), these can be processed asynchronously to reduce the load during peak times.

One challenge we faced in dealing with our dataset is Query Performance. We implemented indexing concept to our dataset to improve performance time. Here we are explaining the use of indexing in the queries

For instance, let’s consider query that retrieves monthly spending summaries for a specific student and year from the student_spending table without relying on an index or materialized view. This query calculates the total amount spent per category for each month directly from the student_spending table.

See the below query without indexing.

SELECT StudentID, CategoryID, Month, Year,
SUM(Amount) AS TotalSpent FROM student_spending

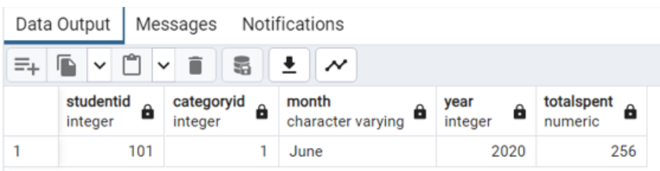
WHERE StudentID = 101 AND Year = 2020 GROUP BY

StudentID, CategoryID, Month, Year

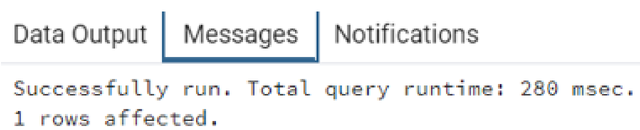
ORDER BY

Month;

Output:

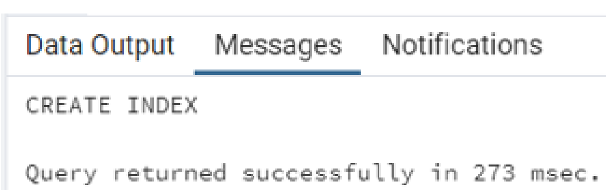


	studentid integer	categoryid integer	month character varying	year integer	totalspent numeric
1	101	1	June	2020	256



See the below query with indexing.

CREATE INDEX idx_student_spending_summary ON student_spending (StudentID, Year, CategoryID, Month);



SELECT StudentID, CategoryID, Month, Year,
SUM(Amount) AS TotalSpent FROM student_spending

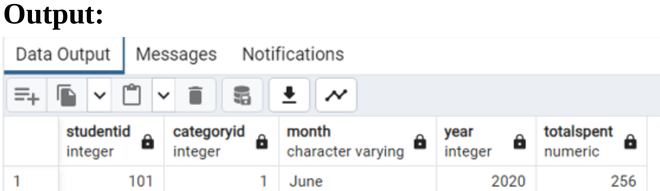
WHERE StudentID = 101 AND Year = 2020 GROUP BY

StudentID, CategoryID, Month, Year

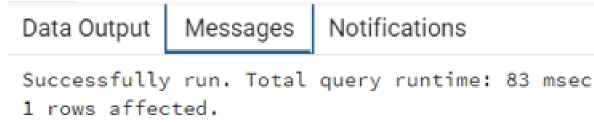
ORDER BY

Month;

Output:



	studentid integer	categoryid integer	month character varying	year integer	totalspent numeric
1	101	1	June	2020	256



Observation:

As we can see clearly that the total query time decreased from 280msec to 83msec

Before Indexing

Full Table Scan: Without the index, the database engine likely performs a full table scan to find the rows matching the WHERE clause conditions. This means every row in the table is checked to see if it meets the criteria (StudentID = 101 and Year = 2020), which is computationally expensive and slow, particularly as the size of the table grows.

Grouping and Sorting Overhead: After filtering, the database must group and sort the results manually based on the CategoryID and Month. This operation requires additional computational resources as the database must sort through all the retrieved data.

Higher Response Time: The overall response time for the query is higher, especially as the dataset grows larger, due to the full table scans and the subsequent need to group and sort a potentially large amount of data manually.

After Indexing

Index Scan or Index Seek: With the composite index in place, the database engine can perform an index scan or index seek, which is much faster than a full table scan. An index seek, in particular, directly finds the rows that match the WHERE clause conditions without scanning unnecessary data.

Efficient Grouping and Sorting: Since the data is already sorted in the order of the index (by StudentID, Year, CategoryID, Month), the database can more efficiently perform the grouping and sorting operations. This pre-sorting by the index reduces computational overhead.

Reduced Response Time: The response time for the query is significantly reduced. This is because the amount of data that needs to be processed is minimized, and the operations of filtering, grouping, and sorting are much more efficient.

Optimized Resource Usage: The use of indexing generally results in better utilization of resources like CPU and memory, because the database engine spends less time processing each query.

So finally, Indexing significantly improves query performance in terms of speed and efficiency.

Task-6

Query 1:

INSERT INTO students (studentid, age, gender, yearinschool, major)
VALUES ('5501', 20, 'Male', 'Sophomore', 'Computer Science');

Data Output Messages Notifications

```
Successfully run. Total query runtime: 256 msec.
5501 rows affected.
```

Output:

Data Output Messages Notifications

	studentid [PK] integer	age character varying (40)	gender character varying (40)	yearinschool character varying (40)	major character varying (40)
5493	5494	22	Female	Freshman	Arts
5494	5495	18	Female	Sophomore	Business
5495	5496	19	Other	Sophomore	Science
5496	5497	18	Other	Senior	Engineering
5497	5498	18	Male	Sophomore	Science
5498	5499	23	Female	Senior	Science
5499	5500	20	Male	Sophomore	Science
5500	7	19	Female	Senior	Engineering
5501	5501	20	Male	Sophomore	Computer Science

Total rows: 5501 of 5501 Query complete 00:00:00.256

Query 2:

INSERT INTO Payment_Methods (MethodID, MethodName) VALUES (6, 'Apply Pay');

Output:

Data Output Messages Notifications

	methodid [PK] integer	methodname character varying
1	1	Credit Card
2	2	Debit Card
3	3	PayPal
4	4	Check
5	5	Wire Transfer
6	6	Apply Pay
7	7	Swift Transfer

Data Output Messages Notifications

```
INSERT 0 1

Query returned successfully in 31 msec.
```

Query 3:

DELETE FROM "Students" WHERE "StudentID" = '5501';

Output:

Data Output Messages Notifications

	studentid [PK] integer	age character varying (40)	gender character varying (40)	yearinschool character varying (40)	major character varying (40)
5492	5493	22	Male	Senior	Arts
5493	5494	22	Female	Freshman	Arts
5494	5495	18	Female	Sophomore	Business
5495	5496	19	Other	Sophomore	Science
5496	5497	18	Other	Senior	Engineering
5497	5498	18	Male	Sophomore	Science
5498	5499	23	Female	Senior	Science
5499	5500	20	Male	Sophomore	Science
5500	7	19	Female	Senior	Engineering

Data Output Messages Notifications

```
DELETE 1

Query returned successfully in 52 msec.
```

Query 4:

Data Output Messages Notifications

	studentid [PK] integer	age character varying (40)	gender character varying (40)	yearinschool character varying (40)	major character varying (40)
17	21	19	Male	Freshman	Science
18	22	23	Female	Senior	Engineering
19	23	19	Female	Junior	Science
20	24	23	Male	Senior	Science
21	25	18	Female	Junior	Engineering
22	26	19	Male	Junior	Business
23	27	22	Male	Junior	Engineering
24	28	21	Female	Freshman	Arts
25	29	18	Female	Sophomore	Business

UPDATE "Students" SET "Major" = 'Engineering' WHERE "StudentID" = 23;

Output:

Data Output Messages Notifications

	studentid [PK] integer	age character varying (40)	gender character varying (40)	yearinschool character varying (40)	major character varying (40)
21	21	19	Male	Freshman	Science
22	22	23	Female	Senior	Engineering
23	23	19	Female	Junior	Engineering
24	24	23	Male	Senior	Science
25	25	18	Female	Junior	Engineering
26	26	19	Male	Junior	Business
27	27	22	Male	Junior	Engineering
28	28	21	Female	Freshman	Arts
29	29	18	Female	Sophomore	Business

Data Output Messages Notifications

```
UPDATE 1

Query returned successfully in 54 msec.
```

Query 5:

SELECT s.StudentID, s.Age, s.Gender, pm.MethodName
FROM Students s
LEFT JOIN Payment_preferences pp ON s.StudentID = pp.StudentID
LEFT JOIN Payment_methods pm ON pp.MethodID = pm.MethodID;

Output:

Data Output Messages Notifications

	studentid integer	age character varying (40)	gender character varying (40)	methodname character varying
1	2980	24	Male	Wire Transfer
2	79	21	Other	Wire Transfer
3	413	21	Female	PayPal
4	3107	24	Female	Wire Transfer
5	1236	20	Female	Debit Card
6	2374	21	Other	Check
7	4116	20	Other	Check
8	5320	24	Female	Credit Card
9	3639	21	Female	PayPal
10	4935	24	Other	PayPal
11	5217	24	Female	Debit Card

```
Successfully run. Total query
runtime: 56 msec.
6988 rows affected.
```

Query 6:

SELECT s.StudentID, SUM(mi.MonthlyIncome) AS TotalIncome
FROM Students s
INNER JOIN Monthly_income mi ON s.StudentID = mi.StudentID
GROUP BY s.StudentID;

Output:

Data Output Messages Notifications

	studentid [PK] integer	totalincome numeric
1	1489	5089
2	4790	1491
3	273	1258
4	2574	2606
5	951	2623
6	2196	682
7	176	717
8	4993	3217
9	4976	3123
10	1003	881
11	5478	2439

```
Successfully run. Total query
runtime: 61 msec.
3500 rows affected.
```

Query 7:

SELECT s.Gender, AVG(mi.MonthlyIncome) AS Avg_Income
FROM Students s
INNER JOIN Monthly_income mi ON s.StudentID = mi.StudentID
GROUP BY s.Gender;

Output:

Data Output Messages Notifications

	gender character varying (40)	avg_income numeric
1	Other	1244.7312390924956370
2	Male	1259.9860440150295223
3	Female	1239.4817518248175182

```
Successfully run. Total query
runtime: 50 msec.
3 rows were affected.
```

Query 8:

SELECT s.StudentID, SUM(sp.Amount) AS Total_Spending
FROM Students s
INNER JOIN Students_Spending sp ON s.StudentID = sp.StudentID
GROUP BY s.StudentID
ORDER BY Total_Spending DESC
LIMIT 5;

Output:

Data Output Messages Notifications

	studentid [PK] integer	total_spending numeric
1	1537	3661
2	1680	3655
3	3967	3613
4	423	3517
5	4347	3385

```
Successfully run. Total query
runtime: 430 msec.
5 rows affected.
```

Query 9:

SELECT s.StudentID, s.Age, s.Gender, a.AcademicYear, sc.ScholarshipName, g.GrantName, l.LoanProvider
FROM Students s
LEFT JOIN FinancialAid_Records f ON s.StudentID = f.StudentID
LEFT JOIN Academic_years a ON f.AcademicYearID = a.AcademicYearID
LEFT JOIN Scholarships sc ON f.ScholarshipID = sc.ScholarshipID

LEFT JOIN Grants g ON f.GrantID = g.GrantID
LEFT JOIN Loans l ON f.LoanID = l.LoanID;

Output:

	studentid integer	age character varying (40)	gender character varying (40)	academicyear character varying (40)	scholarshipname character varying (50)	grantname character varying	loanprovider character varying
1	4476	23	Other	2023-2024	[null]	[null]	[null]
2	3706	18	Other	2020-2021	Merit-Based	Federal Pell	Federal Loan
3	1067	21	Male	2023-2024	Artistic	[null]	[null]
4	1535	23	Female	2023-2024	[null]	Federal Pell	Federal Loan
5	3838	21	Female	2022-2023	[null]	Federal Pell	[null]
6	1214	20	Female	2022-2023	Merit-Based	[null]	Federal Loan
7	5052	18	Other	2022-2023	Merit-Based	[null]	[null]
8	3202	21	Male	2020-2021	[null]	[null]	State Loan
9	3705	24	Male	2021-2022	Academic	[null]	[null]

```
Successfully run. Total query
runtime: 81 msec.
7519 rows affected.
```

Query 10:

SELECT s.StudentID, s.Age, s.Gender, a.AcademicYear, sc.ScholarshipName, g.GrantName, l.LoanProvider
FROM Students s
LEFT JOIN FinancialAid_Records f ON s.StudentID = f.StudentID
LEFT JOIN Academic_years a ON f.AcademicYearID = a.AcademicYearID
LEFT JOIN Scholarships sc ON f.ScholarshipID = sc.ScholarshipID
LEFT JOIN Grants g ON f.GrantID = g.GrantID
LEFT JOIN Loans l ON f.LoanID = l.LoanID
WHERE CAST(s.Age AS INTEGER) BETWEEN 18 AND 22 AND (sc.ScholarshipAmount > 1000 OR g.GrantAmount > 5000);

Output:

	studentid integer	age character varying (40)	gender character varying (40)	academicyear character varying (40)	scholarshipname character varying (50)	grantname character varying	loanprovider character varying
1	3706	18	Other	2020-2021	Merit-Based	Federal Pell	Federal Loan
2	1067	21	Male	2023-2024	Artistic	[null]	[null]
3	1214	20	Female	2022-2023	Merit-Based	[null]	Federal Loan
4	5052	18	Other	2022-2023	Merit-Based	[null]	[null]
5	2216	21	Male	2021-2022	Athletic	[null]	Private Loan
6	307	20	Other	2022-2023	Merit-Based	[null]	[null]
7	3855	19	Other	2023-2024	Academic	Private Grant	[null]
8	5118	18	Male	2023-2024	Artistic	[null]	[null]
9	5447	21	Other	2022-2023	Athletic	[null]	[null]
10	2865	22	Female	2023-2024	Artistic	Federal Pell	[null]
11	1511	21	Other	2021-2022	Merit-Based	[null]	Subsidized Loan

```
Successfully run. Total query
runtime: 48 msec.
1556 rows affected.
```

Query 11:

SELECT s.StudentID,
    SUM(COALESCE(ss.Amount, 0)) AS TotalSpending,
    SUM(COALESCE(g.GrantAmount, 0)) AS TotalGrants,
    SUM(COALESCE(sc.ScholarshipAmount, 0)) AS TotalScholarships,
    SUM(COALESCE(l.LoanAmount, 0)) AS TotalLoans,
    SUM(COALESCE(ss.Amount, 0))
    - SUM(COALESCE(g.GrantAmount, 0))
    - SUM(COALESCE(sc.ScholarshipAmount, 0))
    + SUM(COALESCE(l.LoanAmount, 0)) AS NetTotal
FROM Students s
LEFT JOIN Students_spending ss ON s.StudentID = ss.StudentID
LEFT JOIN FinancialAid_Records f ON s.StudentID = f.StudentID
LEFT JOIN Scholarships sc ON f.ScholarshipID = sc.ScholarshipID
LEFT JOIN Grants g ON f.GrantID = g.GrantID
LEFT JOIN Loans l ON f.LoanID = l.LoanID
GROUP BY s.StudentID;

Output:

	studentid [PK] integer	totalspending numeric	totalgrants numeric	totalscholarships numeric	totalloans numeric	nettotal numeric
83	4634	2226	0	0	0	2226
84	769	0	1341	1487	0	-2828
85	266	533	3602	0	0	-3069
86	2543	0	0	0	0	0
87	1550	0	3343	0	0	-3343
88	1287	0	2404	3340	27555	21811
89	4669	252	0	1487	0	-1235
90	3855	0	1341	4916	0	-6257
91	3840	0	0	0	0	0
92	5461	318	0	0	0	318
93	366	0	0	0	0	0

```
Successfully run. Total query
runtime: 85 msec.
5500 rows affected.
```

Task 7

We'll use the FinancialAid_Records table, which contain records of different types of financial aid each student receives per academic year. We'll identify three problematic queries targeting this table and suggest different optimization techniques for each.

Query 1: Suboptimal Use of Subqueries

SELECT DISTINCT StudentID
FROM FinancialAid_Records
WHERE AcademicYearID IN (SELECT AcademicYearID FROM Academic_years WHERE AcademicYear = '2021-2022');

Data Output | Messages | Notifications

```
Successfully run. Total query runtime: 529 msec.
1198 rows affected.
```

Problem: This query uses a subquery in the WHERE clause to filter FinancialAid_Records based on the AcademicYear. Subqueries can be inefficient, especially if they are not properly indexed, causing repeated execution.

The EXPLAIN Query states as below

EXPLAIN SELECT DISTINCT StudentID
FROM FinancialAid_Records
WHERE AcademicYearID IN (SELECT AcademicYearID FROM Academic_years WHERE AcademicYear = '2021-2022');

Data Output Messages Notifications

	QUERY PLAN text
1	HashAggregate (cost=122.23..135.98 rows=1375 width=4)
2	Group Key: financialaid_records.studentid
3	-> Hash Join (cost=1.06..118.80 rows=1375 width=4)
4	Hash Cond: (financialaid_records.academicyearid = academic_years.academicyear…
5	-> Seq Scan on financialaid_records (cost=0.00..88.00 rows=5500 width=8)
6	-> Hash (cost=1.05..1.05 rows=1 width=4)
7	-> Seq Scan on academic_years (cost=0.00..1.05 rows=1 width=4)
8	Filter: ((academicyear)::text = '2021-2022'::text)

Seq Scan on FinancialAid_Records: This indicates that PostgreSQL is performing a sequential scan on the FinancialAid_Records table. This is a full table scan, meaning it examines every row to see if it meets the criteria.
Subplan: The part of the plan dealing with the subquery (SELECT AcademicYearID FROM Academic_years WHERE AcademicYear = '2021-2022') usually shows a Seq Scan on the Academic_years table if no index is present. This is executed for each row of the outer query, which can be highly inefficient if the Academic_years table is large.

This type of execution plan is generally inefficient because:

Full Table Scans: Both tables might undergo full table scans if no relevant indexes are available.
Repetitive Execution of Subquery: The subquery for fetching AcademicYearID might be executed repeatedly, one for each row processed in the outer query, leading to significant overhead.

Improvement Plan:

Rewrite using JOIN: Replace the subquery with a JOIN, which can be more efficient as it allows better use of indexes.

SELECT DISTINCT far.StudentID
FROM FinancialAid_Records far
JOIN Academic_years ay ON far.AcademicYearID = ay.AcademicYearID
WHERE ay.AcademicYear = '2021-2022';

Data Output Messages Notifications

```
Successfully run. Total query runtime: 99 msec.
1198 rows affected.
```

EXPLAIN SELECT DISTINCT far.StudentID
FROM FinancialAid_Records far
JOIN Academic_years ay ON far.AcademicYearID =
ay.AcademicYearID
WHERE ay.AcademicYear = '2021-2022';

Data Output Messages Notifications

	QUERY PLAN text
1	HashAggregate (cost=122.23..135.98 rows=1375 width=4)
2	Group Key: far.studentid
3	-> Hash Join (cost=1.06..118.80 rows=1375 width=4)
4	Hash Cond: (far.academicyearid = ay.academicyearid)
5	-> Seq Scan on financialaid_records far (cost=0.00..88.00 rows=5500 width…
6	-> Hash (cost=1.05..1.05 rows=1 width=4)
7	-> Seq Scan on academic_years ay (cost=0.00..1.05 rows=1 width=4)
8	Filter: ((academicyear)::text = '2021-2022'::text)

Hash Join: This operation combines rows from FinancialAid_Records and Academic_years based on the matching AcademicYearID. This type of join is generally more efficient than executing a subquery for each row in FinancialAid_Records.
Index Scan on Academic_years: If an index exists on the AcademicYear column, PostgreSQL can quickly locate the rows for the year '2021-2022'. This significantly reduces the number of rows to join, as the database engine can directly fetch the relevant AcademicYearID values using the index.

This improved execution plan enhances performance by:

Reducing Full Table Scans: By using an index on AcademicYear, the database reduces the need to scan the Academic_years table fully.
Efficient Joins: The hash join is typically faster when joining large sets of data, especially when the join condition can be optimized with an index.
By replacing the subquery with a join and ensuring proper indexing, the database engine can optimize the way data is accessed and combined, leading to faster query execution and reduced load on the database system.

Query 2: Aggregating Financial Aid records

SELECT StudentID, AcademicYearID, COUNT(ScholarshipID) AS TotalScholarships, COUNT(GrantID) AS TotalGrants, COUNT(LoanID) AS TotalLoans
FROM FinancialAid_Records
GROUP BY StudentID, AcademicYearID;

Data Output Messages Notifications

```
Successfully run. Total query runtime: 123 msec.
4891 rows affected.
```

Problem: The query performs a full table scan followed by a costly grouping and aggregation operation without the aid of indexes, resulting in inefficient and slow query execution on large datasets.

The EXPLAIN Query states as below

Data Output | Messages | Notifications

	QUERY PLAN text
1	HashAggregate (cost=156.75..191.56 rows=3481 width=32)
2	Group Key: studentid, academicyearid
3	-> Seq Scan on financialaid_records (cost=0.00..88.00 rows=5500 width=...

Seq Scan on FinancialAid_Records: This output would likely indicate a sequential scan of the FinancialAid_Records table, which means every row is examined. This is inefficient for large datasets.

Group Aggregate: This operation groups results by StudentID and AcademicYearID to calculate counts, which can be resource-intensive without proper indexing.

This execution plan is inefficient because it involves a full table scan followed by a potentially costly grouping and aggregation operation on a large dataset.

Improvement Plan:

Create an index on the columns used for grouping (StudentID and AcademicYearID). This can help speed up the grouping operation significantly.

CREATE INDEX idx_student_academic_year ON FinancialAid_Records(StudentID, AcademicYearID);

Data Output | Messages | Notifications

```
CREATE INDEX

Query returned successfully in 558 msec.
```

SELECT StudentID, AcademicYearID, COUNT(ScholarshipID) AS TotalScholarships, COUNT(GrantID) AS TotalGrants, COUNT(LoanID) AS TotalLoans
FROM FinancialAid_Records
GROUP BY StudentID, AcademicYearID;

Data Output | Messages | Notifications

```
Successfully run. Total query runtime: 114 msec.
4891 rows affected.
```

EXPLAIN SELECT StudentID, AcademicYearID, COUNT(ScholarshipID) AS TotalScholarships, COUNT(GrantID) AS TotalGrants, COUNT(LoanID) AS TotalLoans
FROM FinancialAid_Records
GROUP BY StudentID, AcademicYearID;

Data Output | Messages | Notifications

	QUERY PLAN text
1	HashAggregate (cost=156.75..191.56 rows=3481 width=32)
2	Group Key: studentid, academicyearid
3	-> Seq Scan on financialaid_records (cost=0.00..88.00 rows=5500 width=...

Bitmap Heap Scan on FinancialAid_Records: With the index in place, the database might use a bitmap heap scan combined with the index to efficiently locate and group the records.

Index Scan Using idx_student_academic_year: Ideally, the database will utilize the newly created index to quickly group records by StudentID and AcademicYearID, significantly reducing the processing time for the aggregation.

This improved execution plan should minimize the resource-intensive full table scan and utilize the index to streamline the process of grouping and counting entries, leading to faster and more efficient query performance.

Query 3: Distribution of Financial Aid Types by Academic Year

SELECT ay.AcademicYear, COUNT(far.ScholarshipID) AS Scholarships, COUNT(far.GrantID) AS Grants, COUNT(far.LoanID) AS Loans
FROM FinancialAid_Records far
JOIN Academic_years ay ON far.AcademicYearID = ay.AcademicYearID
GROUP BY ay.AcademicYear
ORDER BY ay.AcademicYear;

Data Output | Messages | Notifications

```
Successfully run. Total query runtime: 341 msec.
4 rows affected.
```

Problem: This query is intended to count how many types of financial aid (scholarships, grants, loans) were granted in each academic year. The problems with this query may include:

Inefficient Joins: If there are no indexes on the AcademicYearID fields in both tables, the database engine may resort to full table scans to perform the join.

Large Aggregations: Counting across potentially large datasets without the aid of indexes on the ScholarshipID, GrantID, and LoanID can be resource-intensive.

Sorting Overhead: Sorting the results by AcademicYear without an indexed column can slow down the query execution.

The EXPLAIN statement states:

Data Output | Messages | Notifications

	QUERY PLAN text
1	Sort (cost=173.90..173.91 rows=4 width=122)
2	Sort Key: ay.academicyear
3	-> HashAggregate (cost=173.82..173.86 rows=4 width=122)
4	Group Key: ay.academicyear
5	-> Hash Join (cost=1.09..118.82 rows=5500 width=110)
6	Hash Cond: (far.academicyearid = ay.academicyearid)
7	-> Seq Scan on financialaid_records far (cost=0.00..88.00 rows=5500 width=...
8	-> Hash (cost=1.04..1.04 rows=4 width=102)
9	-> Seq Scan on academic_years ay (cost=0.00..1.04 rows=4 width=102)

Seq Scan on Academic_years (ay): A sequential scan to fetch each academic year.
Seq Scan on FinancialAid_Records (far): A full table scan on the financial aid records to match each academic year.
Hash Join or Nested Loop Join: Depending on the database system, less efficient join methods might be used because of no indexes.
Group and Sort Operations: The operations to group by AcademicYear and then sort the result could be expensive in terms of computation and memory.

Improvement Plan:

Index Creation: Create indexes on columns involved in joins and where conditions to improve the efficiency of these operations.

CREATE INDEX idx_academicyearid ON Academic_years(AcademicYearID);
CREATE INDEX idx_academicyearid_far ON FinancialAid_Records(AcademicYearID);

Executing Query after creating respective Index:

Data Output | Messages | Notifications

```
Successfully run. Total query runtime: 83 msec.
4 rows affected.
```

The EXPLAIN statement states:

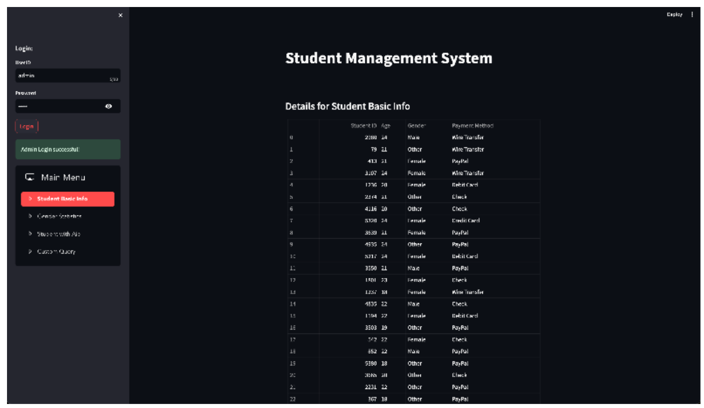
Data Output | Messages | Notifications

	QUERY PLAN text
1	Sort (cost=173.90..173.91 rows=4 width=122)
2	Sort Key: ay.academicyear
3	-> HashAggregate (cost=173.82..173.86 rows=4 width=122)
4	Group Key: ay.academicyear
5	-> Hash Join (cost=1.09..118.82 rows=5500 width=110)
6	Hash Cond: (far.academicyearid = ay.academicyearid)
7	-> Seq Scan on financialaid_records far (cost=0.00..88.00 rows=5500 width=...
8	-> Hash (cost=1.04..1.04 rows=4 width=102)
9	-> Seq Scan on academic_years ay (cost=0.00..1.04 rows=4 width=102)

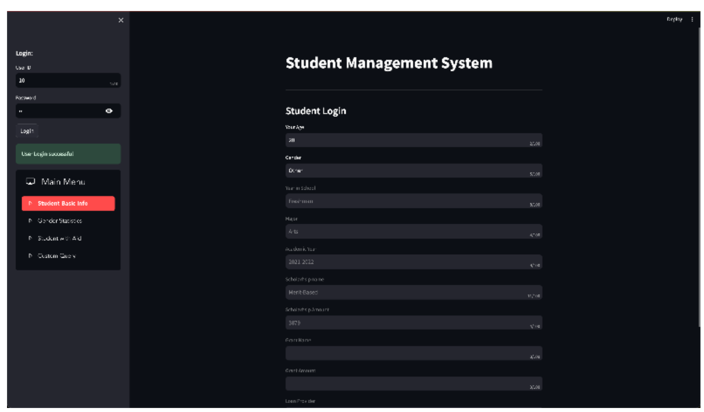
Index Scan on Academic_years (ay): The index on AcademicYearID allows the database to quickly locate the necessary academic years without scanning the entire table.
Index Scan on FinancialAid_Records (far): Similar to Academic_years, an index scan is performed using the new index on FinancialAid_Records, which matches records more efficiently.
Hash Join: With both sides of the join operation being indexed, a hash join might be utilized, which is generally faster and more efficient for larger datasets.
Efficient Grouping and Sorting: The presence of indexes helps to manage the grouping and sorting operations more efficiently.
This improvement approach minimizes the computational overhead and speeds up the execution time significantly, demonstrating the importance of indexing in optimizing complex SQL queries.

Website using the Database:

Admin Access:



Student Access:



Please watch the demo video to fully understand the working of the application.

REFERENCES

Dataset: https://www.kaggle.com/datasets/sumanthnimmagadda/student-spending-dataset?resource=download