# Accounting Software (Team Name: Swifties)

Adarsh Cheeti
*Computer Science and Engineering*
*University at Buffalo*
adarshch@buffalo.edu

Bhanu Nikhil Guntumadugu
*Computer Science and Engineering*
*University at Buffalo*
bhanunik@buffalo.edu

Rahul Nuthalapati
*Computer Science and Engineering*
*University at Buffalo*
rahulnut@buffalo.edu

***Abstract*—The digital accounting in this modern era heavily depends on database systems, the accounting software is built on. This project helps address the needs of small-sized businesses. The database is built on PostgreSQL, with essential relations and dependencies.**

***Keywords— SQL, accounting, book-keeping, order management, inventory, invoicing, taxes***

## I. INTRODUCTION

The small business heavily depend on the book-keeping software in order to manage their inventory, record sales, purchases and their expenses too. And most of the business depend on paper till date to manage their finances and the business transactions. We here present the database built on PostgreSQL with necessary relations and the possible dependencies which makes the work of accounting entries for these small businesses a simple job.

The typical work of small scale businesses include purchasing the inventory from other businesses, and making sales from the inventory that the business has. So the key functionalities here in the database include Invoicing which is otherwise called, Sales, Billing also known as Purchasing, and the respective item level tables for both the Sales and Purchases, and Expenses for the cost incurred for the business, and other relations which will be required by the before detailed relations. In conclusion, this project aims to address the ease of use and accessible accounting for the small scale businesses.

## II. PROBLEM STATEMENT

The current practices of the small businesses for the order management and financial management practices include manual entries by the persons either in a paper or an electronic Excel format. This approach can be error prone, and it is a time-consuming task to record each transaction. It also lacks features like security and control. It is hard when you want to retrieve the required data, for example, you want to retrieve specific customer's transactions. Moreover, both of these methods are not the most efficient form of storing the data of a business. These methods are not only time consuming, but also lead to error prone and becomes a burden to maintain when the business expands as date, volume grows and complexity increases.

So, when a business expands, its accounting needs play a crucial role in maintaining business records. Designing and using a database that is factually based on the business needs of small-scale businesses will vastly improve the financial situation of the business. When the website is built on top of this database, it becomes easy to use, user-friendly software. Businesses can use this database to effectively store, retrieve orders, and inventory, and manage the business with ease. This project leverages a well-designed database for efficient financial record maintenance, order, and transaction management.

## III. BUSINESS NEEDS AND DATABASE DESIGN

The project builds on the essential relations that small businesses function on. The basic business process and the details of the flow of events are as follows:

a. The business owner purchases inventory from another business.
b. This will require raising a bill for the owner, which shows the amount the business owner owes to the other business, called vendor in this case.
c. The inventory purchases will be stored as items, to manage what and how many of the items are present as part of the inventory management.
d. Now, the business owner makes sales to another business/retail customers, the purchasing entity here is called customers.
e. This will require raising an invoice for the buying entity.
f. The inventory must be adjusted accordingly in the inventory management list.
g. The other most important type of transaction for a business is the expense. This gets created when the business incurs an expense as part of its operations.
h. The expenses can be billable and non-billable, if billable, the expense can be converted to an invoice and be issued to a customer, for which the customer owes the amount now.

i. Now, all the transactions below involve owing money from one business to another. The owes due are cleared once payment is made on the transaction issued. Which will clear the existing owes, based on the payment amount.
j. Now, all these need to be maintained in accounts to know the current financial status of the company. So, these transactions are recorded as amounts in General Ledger, which is the critical part of the accounting software.
k. All the transactions here incur taxes during sales and purchases. So, the amounts will be inclusive of the taxes as required by the government rules.

Keeping this flow of events in mind, relations need to be designed with respective needs for each kind of operation. And the main table, General Ledger, must be connected with most of the tables within the database, as it is the one carrying the current balances of the business. So, the relations are designed as follows:

a. The Sales table representing invoices towards the customers, and the Sale Items table representing each item present within that sale i.e., you need to represent all the items present in an invoice transaction separately.
b. In the same way, the Purchase table represents the bills which have been issued by Vendors, and the purchase Item table represents the number and details of items for each entry of purchase table.
c. The Expense table represents the costs incurred during the operation of the business. As mentioned in the business flow, this could sometimes be due to the operations required by the customer, which then can be converted to an invoice in the Sales table.
d. Each of the above tables will be provided with a separate column for storing the taxes amounts.
e. Now, Payments table is added to clear off the owing amounts between each customer and vendors. Though Payments are raised on top of bills or invoices, they can be raised without them, in which case it will be treated as just payment transactions or payment advances.
f. The Bills and Invoices will directly manipulate on the Items table as the sales/purchases are dealt with items.
g. The Customers and Vendors tables are the ones which link each of the above transactions.
h. Now, all of these transactions are stored and maintained on General Ledger in their respective accounts.
i. At the end, there is a user's table, which maintains and manages the different types of users who can manipulate what type of data within the database.

## IV. SCOPE AND EXTENSION

The main aim of this project includes developing and deploying a complete database-driven solution for order and financial management that is suited to the demands of small and medium enterprises. This solution tries to overcome the problems of handling the data manually and Excel-based approaches by implementing an efficient, and user-friendly accounting software.

The key components of the project scope are:
**Database design**: Creating relational database to store and handle order and financial data. This involves defining tables, columns, data types, and the connections between them.
**System Development**: Build a simple and user-friendly database software application that integrates with the database, allowing users to smoothly enter, retrieve, and manage orders, transactions, and financial information.
**Functionality:**
*Order Management:* Implementing functionalities to manage the order lifecycle. This could involve the process of order capturing, tracking, and fulfilling customer orders.
*Financial Management:* Creating features for managing financial transactions. This might entail recording sales, purchases, and payments.
**Security and Access Control**: Implement robust security measures to secure sensitive business data and ensure appropriate access control by giving user control access.

**Extension:** The extension of this project includes adding additional features and optimizing the existing features to improve overall functionalities of the software. Potential improvements may include:
*Mobile application*: Develop mobile application or web application, where user can access and control the software, enhancing accessibility.
*Reporting and Analysis*: Incorporate reporting features to provide financial reports, sales reports, and other analytics to help with taking decisions.
*Tax returns*: The taxes that are calculated for each transaction can be appended to the required format as required by the government of software use and can be used to file tax returns.

## V. TARGET USER

The target users of this accounting software are small, medium scale business owners and their staff who are currently struggling with their order and financial management due to volume of the data. As there is lack of digital data, it is hard to retrieve the needed information too.

Scenario: Organic grocery is a mid-sized grocery store with nearly 30 employees. They decide to use the accounting software to maintain their order and financial management.

IT person within store: Grocery store will have a dedicated person who can manage database. Ideally this person would have beginner database management experience. He would be responsible for:

1. Configuring and setting up the database.

2. Creating user accounts with different user access like (owner, manager, staff).

3. Scheduling the regular backups.

4. Monitoring the performance of the database regularly.

And the next level of user here is the general salesperson within the store. As the database system is accessible via multiple roles to all the employees within the store, it can be accessed by multiple salesperson to sell the items to customers.

Now, as this is a bulk selling business as well as retail serving business, the customers to the business can be both businesses and end customers. And the employees can use the software to sell the required items to customers.

## VI. E/R DIAGRAM

The first important table within the database is, Users table. The table contains different roles and the security for each user to login, to prevent unauthorized data manipulation. The E/R diagram can be seen in the figure below.

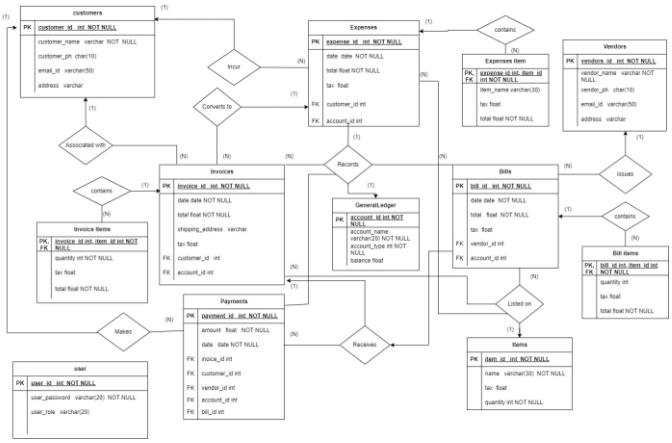


Fig 1

Now, the tables are explained below in the order of the business process of an organization.

1. *Vendors(vendors_id: int NOT NULL, vendor_name: varchar NOT NULL, vendor_ph: char(10), email_id: varchar(50), address: varchar)*

   The Vendors table creates the origin of the business. The vendors are created at the beginning of the business, from whom the inventory can be bought an then stores to sell to the customers in the future. They are used in the future to issue bills to and make payment to.

   The primary key here is *vendors_id: int*, which uniquely identifies each vendor, and can be used as the foreign key in *bills* table and in *payments* table to track the person who had done the transaction.

   The *vendor_name (varchar), vendor_ph (char10), email_id (varchar(50))* and *address* (*varchar*) are used for storing the details of the vendor. And all of these attributes except vendor_name can be set to null as there is no need for particularly storing in the extra details of vendor except name. The tuples here will not be deleted if there are any foreign key references or dependencies.

2. *Customers(customer_id: int NOT NULL, customer_name: varchar, customer_ph: char(10), email_id: varchar(50), address:varchar)*

   The Customers stores the list of customers that the business is planning to sell the items with. The items are sold to customers and they do payments in response as they owe money for items bought.

   The primary key here is *customer_id and it is an int*, it can clearly identify the customer from the id. The *customer_id* acts as foreign key in the invoices and payments table so as to identify the contact (customer/vendor) who made the transaction.

   The remaining attributes are, *customer_name: varchar, customer_ph: char(10), email_id: varchar(50), address:varchar*, within that customer name cannot be null as the name is needed to identify each customer without the confusion of primary key. And the remaining attrributes can be filled null. The tuples here will not be deleted if there are any foreign key references or dependencies.

   **Note:** As both customers and vendors tables are storing phone numbers and mail ID, only one phone number and mail ID can be entered per contact (customer/vendor), so that atomicity can be preserved. And there is no need for multiple email IDs and phone numbers here.

3. *Items(item_id: int, name: varchar(30), tax: float, quantity: int)*

   The items table though having link between the vendors and the customers table, are not related directly in the schema. The items in fact are more related towards the bills and invoices tables, as the transactions include direct mentioning of items within them.

   The primary key here is *item_id:int* , and the same *item_id* can act as foreign key for *invoices*, *invoice_items*, *bills*, *bill_items*, *expenses*, *expense_items* tables.

   The remaining attributes are *name: varchar(30), tax: float, quantity: int.* Within them name, and quantity cannot be null as you need name for identification purposes and quantity to maintain the stock count. And tax is filled if there is tax on the item, else it will be set to default 0. The tuples here will not be deleted if there are any foreign key references or dependencies.

4. *Bills(bills_id: int NOT NULL, bill_date: date, total: int, tax: float, vendor_id: int, account_id: int NOT NULL)*

   Now, we know that bills are issued by vendors to the business as part of buying items from the vendors. So, the tables involved here are Bills, Vendors, Items.

   But one more thing to mention here is that, a bill can

contain multiple items, and all of them cannot be stored within the same bill ID. Therefore, the bill table has to be decomposed to another table in order to accommodate, items within bills. So, another table Bill_items is created.

Now, as for *Bills* table, the primary key is bills_id: int. The bills id acts as primary key here and foreign key inside Bill_items table.

The other attributes include *bill_date: date, bill: int, tax: float* and *vendor_id: int.* The bill date here cannot be null, to know the date the transaction was made. And the total_bill cannot be null too, as amount cannot be empty. tax is not a mandatory field here, and the default value will be 0. *The vendor_id* is foreign key from the *Vendors* table, in order to know from which vendor is the bill issued to. The account_id from GeneralLedger is present here to represent the account in which the transaction should reflect in, it is mandatory as each transaction has to definitely reflect in an account. The tuples here will not be deleted if there are any foreign key references or dependencies.

5. *Bill_items(bill_id: int, item_id: int NOT NULL, quantity int, tax int, total float NOT NULL)*

   Now, the items which are present in the Bills table has to be maintained separately here as part of the Bill_items relation.

   Both the bill_id and item id combined are primary keys here. The bill_id is foreign key from Bills table and item_id is foreign key from items table. They both together forms primary key here as there can be multiple rows with the same bill_id, and adding item id here would make it the primary key.

   The remaining keys here are *quantity int*, *tax: int* and *total float NOT NULL*, The quantity here will be default 1. The tax can be set 0 as default value and the item total here cannot be null as it is amount. The tuples here will not be deleted if there are any foreign key references or dependencies.

6. *Invoices: (invoice_id: int NOT NULL, date: date NOT NULL, total: float NOT NULL, shipping_address: varchar, tax: float, customer_id: int, account_id: int NOT NULL)*

   Invoices relation is created here to store details related to purchase of items which are to be issued to customers. Here, similar to bills relation, an invoice can contain multiple items and all of them cannot be stored with the same invoice_id. Hence, we decompose invoices relation into another table to accommodate items within invoices. Here, the primary key is invoice_id: int. This unique value helps identify each invoice. We have foreign key customer_id which helps us connect invoices relation with customers relation. Other attributes here include date: date, total: float, shipping_address: varchar, tax: float. Date just helps us identify the date on which invoice has been generated. total and tax are the total amount and taxes associated with the purchases in invoice respectively. shipping_address include the address items need to be shipped to. Default value of tax is 0. Attributes other than invoice_id, date and total can be null. The tuples here will not be deleted if there are any foreign key references or dependencies.

7. *Invoice Items(invoice_id: int, item_id: int NOT NULL, quantity: int NOT NULL, tax int, total float NOT NULL)*

   The items which are present in the invoice relation has to be maintained separately here as part of the invoice_items relation. Here, both the invoice_id and item id are combined primary keys. The invoice_id is foreign key from Invoices table and item_id is foreign key from items table. They both together form as primary key here as there can be multiple rows with the same invoice_id. So, adding item_id here as primary key would help us identify multiple items with same invoice id uniquely. The remaining keys here are quantity int, tax: int and total float NOT NULL. Default value of quantity here will be 1 and default value of tax can be set 0. The total here cannot be null as it is amount. The tuples here will not be deleted if there are any foreign key references or dependencies.

8. *Expenses (expense_id: int NOT NULL, date: date NOT NULL, total: float NOT NULL, tax: float, customer_id: int, account_id: int NOT NULL)*

   Expenses relation is created to store the expenses occurred for a own business cause or else during customer's sale for a particular purpose.

   Here, primary key is *expense_id*, which uniquely identifies each set of items purchased for a specific purpose. It is used as foreign key in Expenses item relation to uniquely identify each item involved in expenses.

   Also, *date (date)*, *expense_total (float(20))*, *tax (float(10))* are used to store date during which expenses are made, total amount and taxes incurred for the expense respectively. The date here cannot be null, to know the date the transaction was made. And the expense_total cannot be null too, as amount cannot be empty. tax is not a mandatory field here, and the default value will be 0. *The customer_id* is foreign key from the *Customers* table, if in case the expense is incurred while doing work for a particular customer. This column can be set null as all expenses are not related to customers. The account_id from GeneralLedger is present here to represent the account in which the transaction should reflect in, it is mandatory as each transaction has to definitely reflect in an account. The tuples here will not be deleted if there are any foreign key references or dependencies.

9. *Expenses item (expense_id: int NOT NULL, item_id: int, quantity: int, tax: float, total: float NOT NULL)*

   The Expenses item table here is created as there can be multiple items within the same expense and they cannot be stored within same row in Expense table. Here, those Items will span over multiple rows. *expense_id* as well as *item_id* are combined to form primary key, which helps us uniquely identify each item involved during an expense is made. Here, expense_id and item_id are foreign keys from Expense table and Items table respectively. Here, there can be chance of item being a service, in which case, the item has to be added to the Items table first and then record the expense here.
   Also, quantity, tax, total are used to store quantity associated with each item purchased for an expense, total amount and taxes incurred for the item respectively. All of these attributes except expenses_id, ItemId and total can be set to null. The tuples here will not be deleted if there are any foreign key references or dependencies.

10. *Payments(payment_id: int NOT NULL, amount: int NOT NULL, date date NOT NULL, transaction_id: int, customer_id: int, vendor id: int, account_id: int NOT NULL)*

    The payments table stores the amount transactions from or to either customers or vendors. The payment can be due to sale of an item with an invoice or due to a purchase of an item with a bill from another vendor. Therefore there has be need for including transaction_id in the Payments table. The customer_id or vendor id, either only one them has to be filled.

    The payment_id is the primary key here, as this does not have any dependency with other attributes here and it can clearly define the whole table. And transaction_id, customer id, vendor id are foreign keys from Invoices/Bills/Expenses, Customers, Vendors tables respectively.

    The amount, date cannot be null here, as they are basic attributes of a payment transaction. The transaction id can be null too, as if there is case where payment has been done in advance or to just have a payment itself as a transaction. In this case too customer id, vendor id will be null. The account_id from GeneralLedger is present here to represent the account in which the transaction should reflect in, it is mandatory as each transaction has to definitely reflect in an account. The tuples here will not be deleted if there are any foreign key references or dependencies.

11. *GeneralLedger(account_id int NOT NULL, account_name varchar(20) NOT NULL, account_type int NOT NULL, balance float)*

    The GeneralLedger holds the financial record of the business through its accounts.

    The account_id is the primary key here, which uniquely identifies the account. account_name here holds the name of the account, which cannot be null. account_type hold the type of the account i.e, asset, liability, expense, income and equity. And these are represented using integer, with unique integer for each account. Therefore there would be n rows for n type of accounts in General Ledger. And each account in GeneralLedger holds some amount to it. This is stored inside balance attribute. The default value for balance will be 0, when there are no transactions made into balance. The tuples here will not be deleted if there are any foreign key references or dependencies.

## VII. BCNF

*Customer Table:*
Schema: Customer(customer_id, customer_name, customer_ph, email_id, address)
Functional Dependencies:
Customer_id -> customer_name, customer_ph, email_id, address.
BCNF Status: As customer_id is the only primary key and there are no partial and transitive dependencies. The above relation is in BCNF.

*Expenses Table:*
Schema: Expenses(expenses_id, date, expenses_total, tax, item_id)
BCNF status: As expenses_id is only the primary key and there are no partial and transitive dependencies, the relation is in BCNF.

*User table:*
Schema: user(user_id, user_password, user_role)
Functional Dependencies:
user_id -> user_password, user_role
BCNF status: The above relation has only one primary key and there are no partial and transitive dependencies, the relation is in BCNF.

*Items:*
Schema: items(item_id, name, tax, type)
Functional Dependecies:
Item_id -> name, tax, type
BCNF status: The above has only one primary key and there are no partial and transitive dependencies. The above relation is in BCNF.

*Vendors:*
Schema: vendors(vendor_id, vendor_name, vendor_ph, email_id, address)
Functional Dependecies:
vendor_id -> vendor_name, vendor_ph, email_id, address
BCNF status: The relation has only one primary key and there are there are no partial and transitive dependencies. The above relation is in BCNF.

*Bills:*
Schema: bills(bill_id, date, total_bill, tax, vendor_id, item_id)
BCNF status: The relation has only one primary key (bills_id) and there are there are no partial and transitive dependencies.

The above relation is in BCNF.

*General Ledger:*
Schema: general ledger( account_id, account_name, balance)
Functional Dependencies:
account_id -> account_name, balance
BCNF Status: The relation has only one primary key and there are there are no partial and transitive dependencies. The above relation is in BCNF.

*Invoice Items:*
Schema: Invoice items(invoice_id, item_id, quantity, tax, total)
Functioanl Dependencies:
Invoice_id, item_id -> quantity, tax, total

BCNF Status: Here in this schema two attributes together can be called as primary key (invoice_id, item_id). This combination of attributes is unique in relation. Therefore, no other subset of attributes can functionally determine any other non-prime attribtes, the relation is in BCNF.

*Bill Items:*
Schema: bill items(bill_id, item_id, quantity, tax, total)
Functioanl Dependencies:
Bill_id, item_id -> quantity, tax, total
BCNF Status: Here in this schema two attributes together can be called as primary key (invoice_id, item_id). This combination of attributes is unique in relation. Therefore, no other subset of attributes can functionally determine any other non-prime attributes, the relation is in BCNF.

*Expenses Items:*
Schema: expenses items(expense_id, item_id, quantity, tax, total)
Functioanl Dependencies:
expense_id, item_id -> quantity, tax, total
BCNF Status: Here in this schema two attributes together can be called as primary key (invoice_id, item_id). This combination of attributes is unique in relation. Therefore, no other subset of attributes can functionally determine any other non-prime attributes, the relation is in BCNF.

Phase 2

HANDLING LARGE DATASET

We generated a huge amount of data for our database. So while working on the data, it took prolonged time due to the operations such as Joins, Sum, Avg, Groupby..etc functions.

Because of this, we faced performance and efficiency issues. In order to overcome these issues, we implemented indexing for few of our columns. This helped the database to quickly locate the rows which match certain criteria, and reduce the time needed for query execution.

While executing, a few complex queries took longer time to execute. We looked for patterns involved in queries, like identifying columns involved in slow queries. Then we planned to implement indexing on these columns.

While indexing can improve query performance, over-indexing can also increase overhead on operations like INSERT, UPDATE, etc,. which involves data modification. So, we limited indexing to a few selected columns.

For analysis part, we have observed execution time for few queries with and without indexing just to make sure the perfomance. We were able to execute queries a bit faster when we included indexing.
Here is an example below how much execution time it had taken to fetch data from database without indexing.

This is the query to find the sum of quantities, total sales, and the average tax for each item, grouping by the item's name and filtering for those items with total sales over $1000:

Before Indexing:

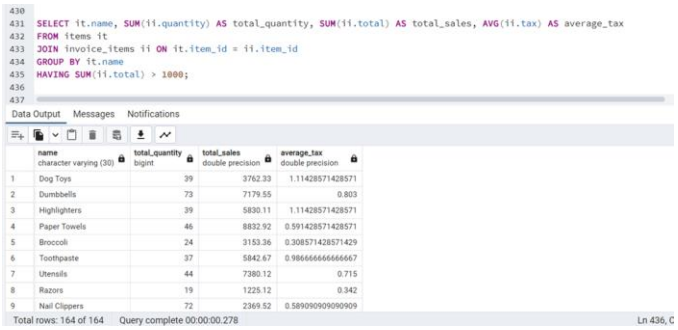

Fig 2

After Indexing :

without indexing it took nearly 0.278 milli seconds. But after using indexing on specific columns (Invoice_items) the execution time got lower. Indexing helps the database to quickly locate the rows which match certain criteria, and reduce the time needed for query execution.

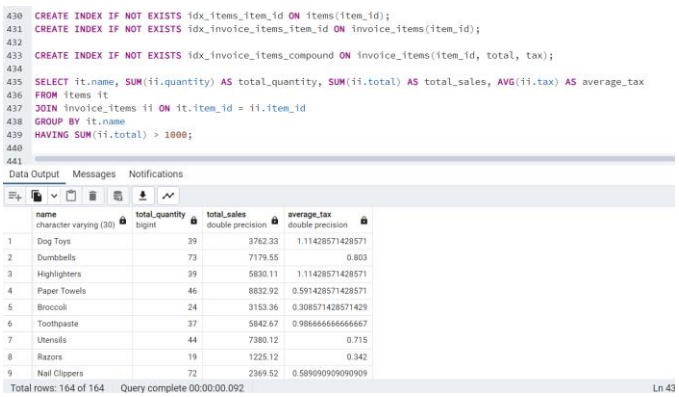

Fig 3

IX. DATASET TESTING WITH SQL QUERIES

1.Query to insert a record into customer table.

```
INSERT INTO customers (customer_id, customer_name, customer_ph, email_id, address)
VALUES (200,'John Doe','900-199-6949','elijah@example.com','74258 heath Square Suite 122 Lake Dylanview, KS 14214');
```

Fig 4

Result:

Data Output | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 55 msec.
```

Fig 5

```
select *from customers where customer_id=200;
```

| customer_id [PK] integer | customer_name character varying | customer_ph character varying (13) | email_id character varying (50) | address character varying |
|---|---|---|---|---|
| 200 | John Doe | 900-199-6949 | elijah@example.com | 74258 heath Square Suite 122 Lake Dylanview, KS 142... |

Fig 6

2. Deleting a record from vendors table.

```
delete from invoices where invoice_id = 100;
```

```
DELETE 1

Query returned successfully in 61 msec.
```

Fig 7

3.Updating the record in vendors table where vendor id=1.

```
UPDATE vendors
SET vendor_name = 'Updated Vendor Name', vendor_ph = '1234567890', email_id = 'update@example.com',
address = 'New Address, City, Country'
WHERE vendor_id = 1;
```

Fig 8

Result:

```
select *from vendors where vendor_id=1;
```

| vendor_id [PK] integer | vendor_name character varying | vendor_ph character varying (13) | email_id character varying (50) | address character varying |
|---|---|---|---|---|
| 1 | Updated Vendor Name | 1234567890 | update@example.com | New Address, City, Country |

Fig 9

4. Query to select all the invoices along with the customer details of those invoices.

```
SELECT i.invoice_id, i.date, i.total, c.customer_name, c.email_id
FROM invoices i
LEFT JOIN customers c ON i.customer_id = c.customer_id
ORDER BY i.date DESC;
```

| invoice_id integer | date date | total double precision | customer_name character varying | email_id character varying (50) |
|---|---|---|---|---|
| 331 | 2024-05-02 | 3729.32 | James Garcia | timothy22@example.com |
| 127 | 2024-05-02 | 561.03 | Johnathan Payne | elizabeth12@example.net |
| 341 | 2024-05-02 | 3289.55 | Bobby Hale | patriciamccoy@example.net |
| 340 | 2024-05-02 | 2076.42 | Joseph Harris | mollycruz@example.com |
| 401 | 2024-05-01 | 600 | Joseph Harris | mollycruz@example.com |
| 201 | 2024-05-01 | 2513.01 | Victoria Robbins | michaelsmith@example.org |
| 101 | 2024-04-29 | 1348.41 | Brian Holloway | rebekahwelch@example.net |
| 329 | 2024-04-27 | 1431.26 | Holly Conley | markanderson@example.org |
| 280 | 2024-04-24 | 1677.08 | Michele Steele | bburnett@example.com |
| 245 | 2024-04-23 | 2081.88 | Billy Lee | bjones@example.net |
| 384 | 2024-04-22 | 1938.36 | Joseph Harris | mollycruz@example.com |
| 164 | 2024-04-22 | 1899.32 | Jacqueline Oneal | hensleysummer@example.net |
| 158 | 2024-04-22 | 1550.89 | Jonathan Hardy | adam33@example.org |
| 281 | 2024-04-21 | 1710.03 | Michael Stout | ugregory@example.net |
| 147 | 2024-04-21 | 874.73 | Jeffrey Kemp | brownelizabeth@example.net |

Fig 10

5. Query to find the maximum invoice amount of each customer using grouby and max.

```
SELECT c.customer_name, MAX(i.total) AS max_invoice_amount
FROM customers c
JOIN invoices i ON c.customer_id = i.customer_id
GROUP BY c.customer_name
ORDER BY max_invoice_amount DESC;
```

| | customer_name character varying | max_invoice_amount double precision |
|---|---|---|
| 1 | Ashley Molina | 5422.42 |
| 2 | Kelsey Booth | 5398.73 |
| 3 | Randy Brooks | 5395.71 |
| 4 | Michael Miller | 5177.12 |
| 5 | Tanya Simon | 4726.28 |
| 6 | Jessica Conrad | 4726.28 |
| 7 | Christopher Ward | 4657.01 |
| 8 | Diana Heath | 4603.49 |
| 9 | Dr. Michael Young ... | 4520.99 |
| 10 | Rhonda Beasley | 4326.04 |
| 11 | Christopher Delacruz | 4299.61 |
| 12 | Anthony Schmidt | 4212.74 |
| 13 | Kristina Wright | 4029.86 |
| 14 | Christine Clark | 4005.23 |
| 15 | James Garcia | 3991.1 |

Total rows: 49 of 49 | Query complete 00:00:00.057 | Ln 370, Col 34

Fig 11

6. Query to find all the vendors and the bills they have issued using right join.

```
SELECT v.vendor_name, b.bill_id, b.date, b.total
FROM vendors v
RIGHT JOIN bills b ON v.vendor_id = b.vendor_id
ORDER BY v.vendor_name, b.date;
```

| | vendor_name character varying | bill_id integer | date date | total double precision |
|---|---|---|---|---|
| 1 | Anderson, Vasquez and Watson | 294 | 2023-05-25 | 1767.96 |
| 2 | Anderson, Vasquez and Watson | 226 | 2023-06-03 | 3133.26 |
| 3 | Anderson, Vasquez and Watson | 6 | 2023-09-13 | 1244.73 |
| 4 | Anderson, Vasquez and Watson | 108 | 2023-11-05 | 436.63 |
| 5 | Anderson, Vasquez and Watson | 101 | 2023-12-13 | 835.84 |
| 6 | Anderson, Vasquez and Watson | 245 | 2023-12-20 | 5383.39 |
| 7 | Bartlett Group | 188 | 2023-05-12 | 1177.26 |
| 8 | Bartlett Group | 317 | 2023-06-03 | 4071.47 |
| 9 | Bartlett Group | 228 | 2023-07-05 | 2776.59 |
| 10 | Bartlett Group | 237 | 2023-09-02 | 881.52 |
| 11 | Bartlett Group | 34 | 2023-09-05 | 799.52 |
| 12 | Bartlett Group | 287 | 2023-12-25 | 3480.76 |
| 13 | Bartlett Group | 311 | 2024-01-02 | 3024.64 |
| 14 | Bartlett Group | 279 | 2024-01-06 | 2991.2 |
| 15 | Bartlett Group | 355 | 2024-02-05 | 1857.79 |

Total rows: 400 of 400 | Query complete 00:00:00.054 | Ln 372, Col 10

Fig 12

7. Query to calculate the total payments done to each vendor.

```
SELECT v.vendor_name, SUM(p.amount) AS total_payments FROM vendors v
JOIN payments p ON v.vendor_id = p.vendor_id
GROUP BY v.vendor_name
ORDER BY total_payments DESC;
```

Data Output | Messages | Notifications

| | vendor_name character varying | total_payments double precision |
|---|---|---|
| 1 | Montoya, Jones and Simpson | 8587.4 |
| 2 | Powell-Garza | 4928.43 |
| 3 | Leonard, Hess and Cameron | 4591.87 |
| 4 | Soto PLC | 4494.94 |
| 5 | Lawrence Inc | 4048.38 |
| 6 | Cook-Singleton | 3838.62 |
| 7 | Updated Vendor Name | 3831.39 |
| 8 | Garcia-Odonnell | 3580.15 |
| 9 | Calhoun Group | 3149.13 |
| 10 | Anderson, Vasquez and Watson | 3113.26 |
| 11 | Bartlett Group | 2991.2 |
| 12 | Thomas-Carrillo | 2667.22 |
| 13 | Brooks, Thomas and Gould | 2577.06 |
| 14 | English, Barr and Aguilar | 2341.57 |
| 15 | King-Vaughn | 2340.15 |

Total rows: 34 of 34 Query complete 00:00:00.388 Ln 378, Col 6

Fig 13

8. Query to Find out the vendors with the all time highest total amount bill amount and their total billing amount.

```
SELECT v.vendor_id, v.vendor_name,
       CASE WHEN total_billed IS NULL THEN 0 ELSE total_billed END AS total_billed
FROM (
    SELECT vendor_id, SUM(total) AS total_billed
    FROM bills
    GROUP BY vendor_id
) AS vendor_bills
RIGHT JOIN vendors v ON v.vendor_id = vendor_bills.vendor_id
ORDER BY total_billed DESC;
```

Data Output | Messages | Notifications

| | vendor_id [PK] integer | vendor_name character varying | total_billed double precision |
|---|---|---|---|
| 1 | 41 | Bartlett Group | 25892.28 |
| 2 | 25 | Coleman, Thornton and Morales | 23759.48 |
| 3 | 43 | Morales LLC | 22956.83 |
| 4 | 32 | Soto PLC | 22303.89 |
| 5 | 7 | Brown Inc | 21989.66 |
| 6 | 1 | Updated Vendor Name | 20508.16 |
| 7 | 23 | Mendoza-Edwards | 20212.05 |
| 8 | 29 | Cook-Singleton | 20065.55 |
| 9 | 50 | Tucker, Fox and Warren | 20051.85 |
| 10 | 22 | Montoya, Jones and Simpson | 19684.68 |
| 11 | 38 | King-Vaughn | 18065.01 |

Successfully run. Total query runtime: 54 msec. 50 rows affected.

Total rows: 50 of 50 Query complete 00:00:00.054 Ln 391, Col 28

Fig 14

9. Query to find the details of the latest bill for each vendor using subquery.

```
SELECT v.vendor_name, b.bill_id, b.date, b.total
FROM vendors v
JOIN bills b ON v.vendor_id = b.vendor_id
WHERE b.date = (SELECT MAX(date) FROM bills b2 WHERE b2.vendor_id = v.vendor_id);
```

Data Output | Messages | Notifications

| | vendor_name character varying | bill_id integer | date date | total double precision |
|---|---|---|---|---|
| 1 | Reid Inc | 1 | 2024-03-28 | 4126.17 |
| 2 | Flynn, Allen and Espinoza | 13 | 2024-04-24 | 765.28 |
| 3 | Wise-Johnson | 32 | 2024-05-02 | 882.9 |
| 4 | Gaines LLC | 45 | 2024-04-16 | 2279.71 |
| 5 | King-Vaughn | 48 | 2024-03-31 | 2551.79 |
| 6 | Garcia Group | 56 | 2024-04-12 | 1403.1 |
| 7 | Montoya, Jones and Simpson | 60 | 2023-12-30 | 4276 |
| 8 | Moss, Taylor and Huffman | 70 | 2024-04-28 | 814.72 |
| 9 | Wright, Dawson and Alexander | 75 | 2024-02-10 | 1885.67 |
| 10 | Morgan PLC | 79 | 2024-04-01 | 1564.56 |
| 11 | Quinn-Smith | 85 | 2024-04-25 | 573.22 |
| 12 | Beard-Willis | 98 | 2024-02-26 | 747.93 |
| 13 | English, Barr and Aguilar | 102 | 2024-03-28 | 1472.9 |
| 14 | Updated Vendor Name | 119 | 2024-03-01 | 1726.32 |

Successfully run. Total query runtime: 50 msec. 50 rows affected.

Total rows: 50 of 50 Query complete 00:00:00.050 Ln 402, Col 1

Fig 15

10.Query to find out which items are never added in any bill.

```
SELECT it.name
FROM items it
WHERE NOT EXISTS (SELECT * FROM bill_items bi WHERE bi.item_id = it.item_id);
```

Data Output | Messages | Notifications

| | name character varying (30) |
|---|---|
| 1 | Shaving Cream |

Fig 16

11. Query to find the total amount spend on each item in expenses.

```
SELECT i.name, SUM(ei.total) AS total_spent
FROM items i
JOIN expense_items ei ON i.item_id = ei.item_id
GROUP BY i.name;
```

Data Output | Messages | Notifications

| | name character varying (30) | total_spent double precision |
|---|---|---|
| 1 | Bolts | 21822.86 |
| 2 | Washers | 31054.71 |
| 3 | Dryers | 41196.5700000001 |
| 4 | Screws | 11289.02 |

Fig 17

12. Finding the vendors with the highest bill amount totally, and also their total billing amount.

```
SELECT v.vendor_id, v.vendor_name,
       CASE WHEN total_billed IS NULL THEN 0 ELSE total_billed END AS total_billed
FROM vendors v
LEFT JOIN (
    SELECT vendor_id, SUM(total) AS total_billed
    FROM bills
    GROUP BY vendor_id
) AS vendor_bills ON v.vendor_id = vendor_bills.vendor_id
ORDER BY total_billed DESC;
```

Data Output | Messages | Notifications

| | vendor_id [PK] integer | vendor_name character varying | total_billed double precision |
|---|---|---|---|
| 1 | 41 | Bartlett Group | 25892.28 |
| 2 | 25 | Coleman, Thornton and Morales | 23759.48 |
| 3 | 43 | Morales LLC | 22956.83 |
| 4 | 32 | Soto PLC | 22303.89 |
| 5 | 7 | Brown Inc | 21989.66 |
| 6 | 1 | Updated Vendor Name | 20508.16 |
| 7 | 23 | Mendoza-Edwards | 20212.05 |
| 8 | 29 | Cook-Singleton | 20065.55 |
| 9 | 50 | Tucker, Fox and Warren | 20051.85 |
| 10 | 22 | Montoya, Jones and Simpson | 19684.68 |

Total rows: 50 of 50 Query complete 00:00:00.358 Ln 420, Col 1

Fig 18

## XI. QUERY EXECUTION ANALYSIS

The query retrieves total amount spent by each vendor on each account from general_ledger, within a specific data range. It will present the results grouped by vendor name and account name.

```
SELECT v.vendor_name, gl.account_name, SUM(b.total) AS total_spent
FROM vendors v
JOIN bills b ON v.vendor_id = b.vendor_id
JOIN general_ledger gl ON b.account_id = gl.account_id
WHERE b.date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY v.vendor_name, gl.account_name;
```

Data Output | Messages | Notifications

```
Successfully run. Total query runtime: 179 msec.
12 rows affected.
```

Fig 19

Here, The the query did not perform better because it has to scan the whole table. These join operations between vendors, bills and general ledger tables and performing sorting and aggregation operations like GROUPBY causes to increase in execution time. So we used indexing on specific columns.

Before indexing, the query was retrieved in 179 ms.

After Indexing:

```
CREATE INDEX idx_bills_vendor_id ON bills(vendor_id);
CREATE INDEX idx_bills_account_id ON bills(account_id);


SELECT v.vendor_name, gl.account_name, SUM(b.total) AS total_spent
FROM vendors v
JOIN bills b ON v.vendor_id = b.vendor_id
JOIN general_ledger gl ON b.account_id = gl.account_id
WHERE b.date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY v.vendor_name, gl.account_name;
```

Data Output | Messages | Notifications

```
Successfully run. Total query runtime: 98 msec.
50 rows affected.
```

Fig 20

we performed indexing on vendor_id and account_id in bills table. This means there is no need of complete table scans, nested loop joins or hash joins.

After indexing, the query was retrieved in 96 ms. So overall execution time is increased by using indexing.

2. SQL query to generates a report by finding the total amount invoiced to each customer, categorized by the account handling each invoice.

```
SELECT c.customer_name, gl.account_name, SUM(i.total) AS total_amount
FROM customers c
JOIN invoices i ON c.customer_id = i.customer_id
JOIN general_ledger gl ON i.account_id = gl.account_id
GROUP BY c.customer_name, gl.account_name
ORDER BY c.customer_name, gl.account_name;
```

Data Output | Messages | Notifications

| | customer_name character varying | account_name character varying (20) | total_amount double precision |
|---|---|---|---|
| 1 | Amanda Miller | income | 11600.89 |
| 2 | Anthony Schmidt | income | 9477.85 |
| 3 | Ashley Molina | income | 9294.12 |

Total rows: 49 of 49 | Query complete 00:00:00.712

Fig 21

The query can be less efficient as there is need for the complete table scans. Another reason can be the less efficient algorithms for the joins like nested loop join and also the aggregation functions are used on a large dataset.

Before indexing, the query was retrieved in 712 ms.

**After Indexing:**

Here, we performed indexing on account_id in invoices table. This means there is no need of full table scans on the invoices table for performing each join operation, nor use nested loop joins or hash joins.

```
CREATE INDEX idx_invoices_account_id ON invoices(account_id);


SELECT c.customer_name, gl.account_name, SUM(i.total) AS total_amount
FROM customers c
JOIN invoices i ON c.customer_id = i.customer_id
JOIN general_ledger gl ON i.account_id = gl.account_id
GROUP BY c.customer_name, gl.account_name
ORDER BY c.customer_name, gl.account_name;
```

Data Output | Messages | Notifications

```
Successfully run. Total query runtime: 148 msec.
49 rows affected.
```

Fig 22

Data Output | Messages | Notifications

| | customer_name character varying | account_name character varying (20) | total_amount double precision |
|---|---|---|---|
| 9 | Christopher Delacruz | income | 15364.98 |
| 10 | Christopher Ward | income | 11519.8 |
| 11 | Daniel Perkins | income | 14284.89 |
| 12 | David Reyes | income | 21562.1 |
| 13 | Diana Heath | income | 15311.78 |
| 14 | Donald Cuevas | income | 11111.6 |
| 15 | Dr. Michael Young MD | income | 12579.02 |
| 16 | Erica Cox | income | 14067.55 |

Total rows: 49 of 49 | Query complete 00:00:00.148

Fig 23

After indexing, the query was retrieved in 148 ms. So it clearly shows that by using indexing the processing load will decrease thereby execution time decreases.

3. This is the query to find the sum of quantities, total sales, and the average tax for each item, grouping by the item's name and filtering for those items with total sales over $1000:

```
SELECT it.name, SUM(ii.quantity) AS total_quantity, SUM(ii.total) AS total_sales, AVG(ii.tax) AS average_tax
FROM items it
JOIN invoice_items ii ON it.item_id = ii.item_id
GROUP BY it.name
HAVING SUM(ii.total) > 1000;
```

Data Output | Messages | Notifications

| | name character varying (30) | total_quantity bigint | total_sales double precision | average_tax double precision |
|---|---|---|---|---|
| 1 | Dog Toys | 39 | 3762.33 | 1.11428571428571 |
| 2 | Dumbbells | 73 | 7179.55 | 0.803 |
| 3 | Highlighters | 39 | 5830.11 | 1.11428571428571 |
| 4 | Paper Towels | 46 | 8832.92 | 0.591428571428571 |
| 5 | Broccoli | 24 | 3153.36 | 0.308571428571429 |
| 6 | Toothpaste | 37 | 5842.67 | 0.986666666666667 |
| 7 | Utensils | 44 | 7380.12 | 0.715 |
| 8 | Razors | 19 | 1225.12 | 0.342 |
| 9 | Nail Clippers | 72 | 2369.52 | 0.589090909090909 |

Total rows: 164 of 164 Query complete 00:00:00.278 Ln 436, C

Fig 24

**After Indexing**:

Without indexing it took nearly 0.278 milli seconds. But after using indexing on invoice_items (item_id, total, tax) the execution time got lower to 0.092 ms.

```
CREATE INDEX IF NOT EXISTS idx_items_item_id ON items(item_id);
CREATE INDEX IF NOT EXISTS idx_invoice_items_item_id ON invoice_items(item_id);

CREATE INDEX IF NOT EXISTS idx_invoice_items_compound ON invoice_items(item_id, total, tax);

SELECT it.name, SUM(ii.quantity) AS total_quantity, SUM(ii.total) AS total_sales, AVG(ii.tax) AS average_tax
FROM items it
JOIN invoice_items ii ON it.item_id = ii.item_id
GROUP BY it.name
HAVING SUM(ii.total) > 1000;
```

Data Output | Messages | Notifications

| | name character varying (30) | total_quantity bigint | total_sales double precision | average_tax double precision |
|---|---|---|---|---|
| 1 | Dog Toys | 39 | 3762.33 | 1.11428571428571 |
| 2 | Dumbbells | 73 | 7179.55 | 0.803 |
| 3 | Highlighters | 39 | 5830.11 | 1.11428571428571 |
| 4 | Paper Towels | 46 | 8832.92 | 0.591428571428571 |
| 5 | Broccoli | 24 | 3153.36 | 0.308571428571429 |
| 6 | Toothpaste | 37 | 5842.67 | 0.986666666666667 |
| 7 | Utensils | 44 | 7380.12 | 0.715 |
| 8 | Razors | 19 | 1225.12 | 0.342 |
| 9 | Nail Clippers | 72 | 2369.52 | 0.589090909090909 |

Total rows: 164 of 164 Query complete 00:00:00.092 Ln 43

Fig 25

XI. WEBSITE


This is the login page.


Fig 26

Once we login, we have an option to enter any query we would like to execute.

Or we have an option to enter the table name we would like to view.


Fig 27

Here on the above page, we can see the data retrieved by executing SELECT * query on expense_items table.

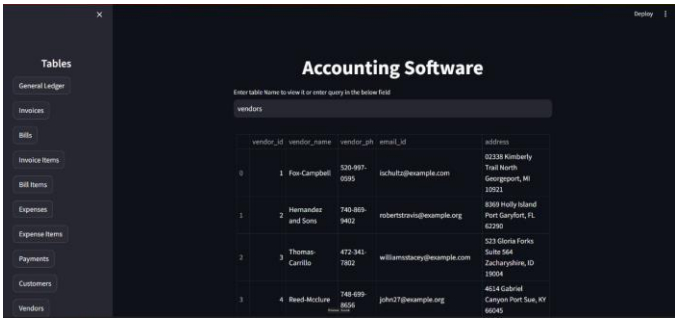


Fig 28

Here on the above, we can see the vendors table being retrieved by entering the table name.

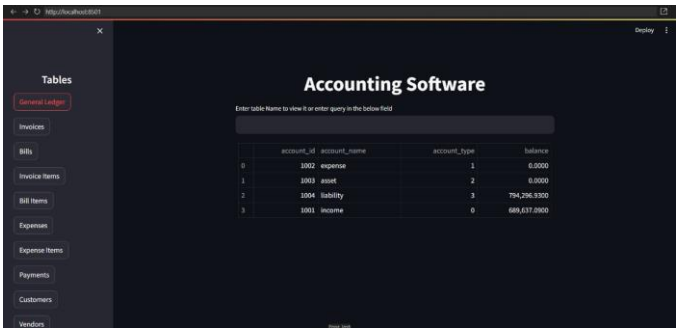


Fig 29

On the side, we also have a display of all the tables in the database. We can simply click on the table we would like to view.

## XII. DATASETS

We are using Sales data from [5] for getting the invoices, invoice items and customers data, which will cover the sales part of the database. And we will combine [5] with [6] for getting their General Ledger transactions. And we use [7] for expense related transactions, which can also be applied for bills if a vendor has been assigned to them. And the vendor names/details will be taken from [8]. Right now we are using these datasets to our use, as different tables serve different purposes and these datasets can be used combined for generating datasets for whole database. We might consider using custom generated datasets from them, if the exact use cases or required columns do no match at the end.

PHASE 2 Situation: We came across the faker library in python which can be used to generate random content using the library functions. We created a python file to run the generation programs. We first generated the rows for the non-dependent tables like customers, vendors, and items. Then the major tables are invoices, invoice_items, bills, bill_items, expenses, and expense_items. Now, the way the data is generated for these tables is that first we get the existing data from already generated tables and use randint and randomly generated the dummy values. Once all are completed it gave more than 5000 rows for all the tables combined.

## REFERENCES

[1] https://www.accountingcoach.com/accounting-basics/explanation.
[2] Course slides of CSE 560 for ER Diagram, BCNF and relation references.
[3] https://www.geeksforgeeks.org/introduction-of-er-model/
[4] https://en.wikipedia.org/wiki/Boyce%E2%80%93Codd_normal_form
[5] https://www.kaggle.com/datasets/dataceo/sales-and-customer-data?select=sales_data.csv
[6] https://www.kaggle.com/datasets/jazidesigns/financial-accounting
[7] https://www.kaggle.com/datasets/bukolafatunde/personal-finance?select=personal_transactions.csv
[8] https://www.kaggle.com/datasets/trobacker/ashevillevendors