# Domain Knowledge

This project develops a system designed to monitor how accountants handle financial records across multiple companies. It meticulously tracks invoice entries and device usage to identify errors or unusual patterns—such as missing invoices, entries logged at odd hours, or excessive reliance on generic expense categories—and provides suggestions for improvement. The system systematically logs comprehensive data, encompassing company details, accountant activities, invoice numbers, transaction specifics, expense types, devices involved, and any detected errors. Through detailed analysis, it uncovers habits like frequent invoice splitting or the use of vague labeling.

Leveraging advanced technology, the system differentiates between effective and poor bookkeeping practices while highlighting error trends. Beyond merely detecting anomalies, it offers actionable insights—recommending optimal times for recording transactions or proper expense categorization—to elevate accounting standards. The outcome is a significant reduction in errors and the adoption of smarter bookkeeping habits. Key features include **tracking companies, accountants, invoices, and devices; logging transactions and errors for in-depth analysis; detecting irregularities like gaps in invoice sequences or delayed entries; and enhancing accuracy through data-driven insights.** Ultimately, it serves as an oversight tool that not only stores financial data but also refines accounting processes by pinpointing issues and fostering improved practices.

# Entity and Relationship Description

Companies and Users

Every company using the system has its own profile—think name, industry, and location. Each one employs a bunch of users, mostly accountants, but sometimes managers too. These users have details like their name, role (say, "Accountant" or "Accounting Manager"), and how long they’ve been at it. Each user is tied to one company and handles tasks like recording invoices. They get assigned a range of invoice numbers to work with, and they’re the ones creating invoices and transaction entries. Mistakes they make get logged as errors, and their actions on machines—like scanning or printing—get tracked too.

Invoice Sequences

Users are given a set of invoice numbers, like 1001 to 1010, to use for a specific period, maybe a year. Each sequence has a start number, end number, and year. A user might have several sequences over time. These ranges help create invoices, and if a number’s skipped, it’s a clue an invoice might be missing.

Invoices

An invoice is basically a financial bill with a unique number and date. It’s tied to a user’s sequence, linking it to that user and their company. Sometimes, an invoice gets split into multiple parts—like different expense categories (think "split booking"). The system checks for gaps in the sequence to catch missing invoices. If something’s off, like a duplicate, it might get flagged with an error log. Users enter these invoices, and in a fancier setup, errors can tie directly to specific invoices.

Transaction Records

These are the nitty-gritty accounting entries—like splitting a $500 invoice into $300 for office supplies and $200 for travel. Each entry has an amount and a timestamp. They’re part of a single invoice and get sorted into expense categories. The system also notes which machine (like a computer or scanner) was used to enter them. This lets you spot patterns, like late-night entries or frequent splits, and catch errors if the invoice total doesn’t match its parts.

Account Categories

This is a list of expense types—travel, utilities, office supplies, you name it. Every transaction gets tagged with one category. It’s handy for seeing if someone’s dumping too much into "Miscellaneous," which might mean they’re not sorting things right. Each category just needs a name, and it keeps transactions organized and summaries clean.

Machines

Machines are the devices—like computers or scanners—used in the process. They’ve got a model and type, say "Dell OptiPlex 7080" for a desktop or "Fujitsu ScanSnap iX1500" for a scanner. Each belongs to a company, and while they’re not locked to one user, the system tracks who uses them. Machines log transaction entries and their own usage records, helping figure out if certain models cause delays or errors.

Machine Logs

Whenever a machine does something big, like scanning an invoice, it gets logged with the operation type (e.g., "Scan") and time. The log ties to the machine, the user, and, if relevant, the invoice number. This catches stuff like an invoice scanned but not recorded—a red flag. It’s great for tracking who’s using what and when, and cross-checking with invoices.

Error Logs

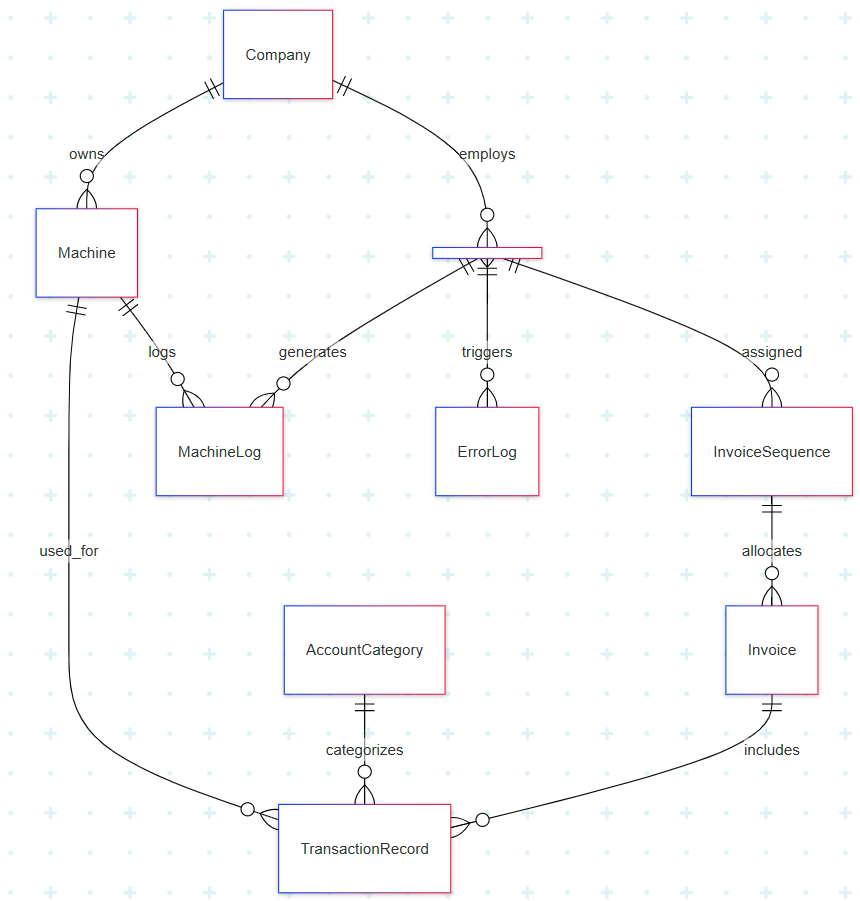
This is where mistakes get noted—things like missing invoices, duplicates, or entries made way too late. Each error has a type (like "LateEntry"), a description, and a timestamp, and it’s pinned to the user responsible. Some errors link to a specific invoice, others don’t. It covers issues like odd-hour entries or mismatched totals. Checking these logs shows who might need a refresher or what problems pop up most.

How It Ties Together

A company has users and machines. Users handle invoices through their sequences, and each invoice breaks into transaction records. Machines and users team up to log usage, while transactions get sorted into categories and tied to machines and invoices. Errors stick to users and sometimes invoices. It’s all connected to keep things running smoothly.

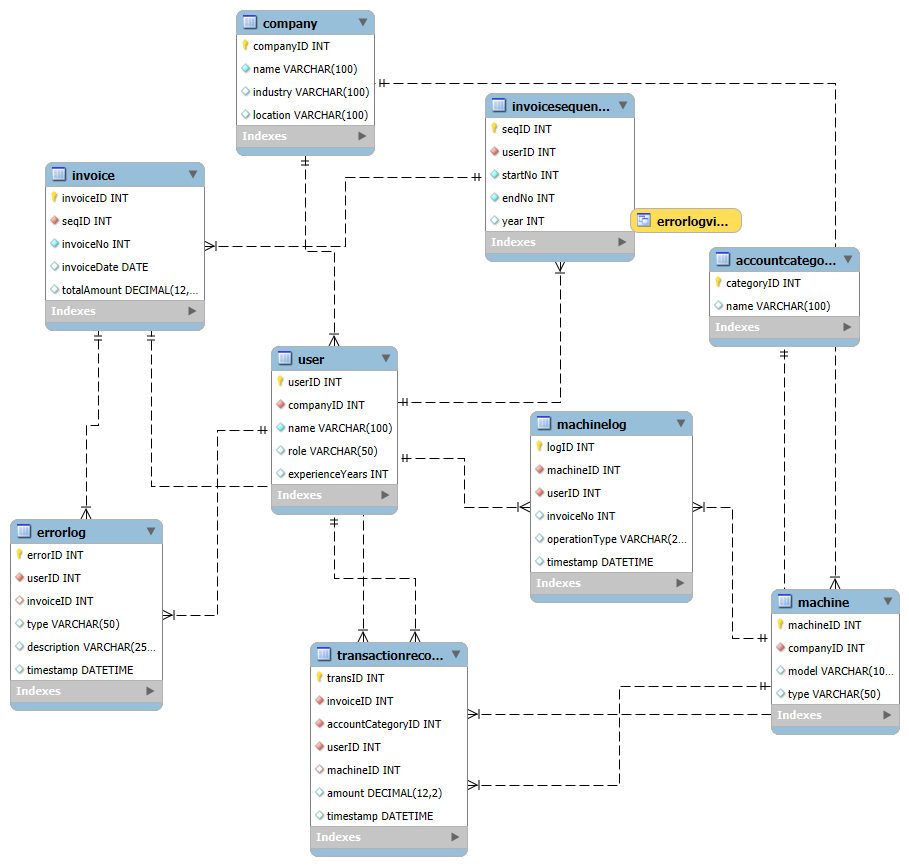
In summary, A Company has many Users and Machines. A User records many Invoices (via their sequences) and each Invoice comprises one or more TransactionRecords. Users and Machines together generate MachineLogs. Each TransactionRecord is categorized by an AccountCategory and is associated with one Machine and one Invoice. Each ErrorLog is attributed to a User (and possibly linked to an Invoice). These relationships are illustrated in the ER diagram below.

# ER Diagram



Note: We use Mermaid software to Draw the ER diagram.

# EER Diagram



# DDL of tables, views, functions procedures

Below we present the SQL DDL statements for creating the tables, along with a view and stored program examples. Primary keys, foreign keys, and any relevant constraints are included, and each is followed by an explanation:

CREATE TABLE Company (

companyID INT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

industry VARCHAR(100),

location VARCHAR(100)

);

Explanation: The Company table holds companies using the system. companyID is the primary key. The name is required. This table is referenced by User and Machine tables (each user and machine is associated with a company).

CREATE TABLE User (

userID INT PRIMARY KEY,

companyID INT NOT NULL,

name VARCHAR(100) NOT NULL,

role VARCHAR(50),

experienceYears INT,

FOREIGN KEY (companyID) REFERENCES Company(companyID)

);

Explanation: The User table stores accountants (and managers). It has a foreign key companyID linking the user to their Company (each user must belong to a company). The primary key is userID. The role field can distinguish subtypes (e.g. "Accountant" vs "Accounting Manager") – in the EER we treat these as sub-classes. experienceYears is a sample attribute that could be used to correlate experience with error occurrence. In this schema, we enforce that companyID must exist in Company (referential integrity).

CREATE TABLE Machine (

machineID INT PRIMARY KEY,

companyID INT NOT NULL,

model VARCHAR(100),

type VARCHAR(50),

FOREIGN KEY (companyID) REFERENCES Company(companyID)

);

Explanation: The Machine table represents devices. Each machine is associated with a Company via foreign key. (If we had a direct assignment to a user, we could also have a userID FK here, but we chose to log usage per user instead.) model and type describe the machine. Machines are referenced by TransactionRecord (which machine used for entry) and MachineLog.

CREATE TABLE AccountCategory (

categoryID INT PRIMARY KEY,

name VARCHAR(100) UNIQUE

);

Explanation: AccountCategory is a lookup for expense/ledger categories (e.g. Travel, Office Supplies). The category name is unique for easy lookup. This table is referenced by TransactionRecord to enforce valid category entries. (No FK dependencies here beyond TransactionRecord referencing it.)

CREATE TABLE InvoiceSequence (

seqID INT PRIMARY KEY,

userID INT NOT NULL,

startNo INT NOT NULL,

endNo INT NOT NULL,

year INT,

FOREIGN KEY (userID) REFERENCES User(userID)

);

Explanation: InvoiceSequence defines a range of invoice numbers allocated to a user (e.g. user 1 might have startNo=1001, endNo=1010 for year=2024). It has a foreign key to User (the user who owns this sequence). The combination of (userID, startNo, endNo) should ideally be unique and non-overlapping; here we assume the input is managed so that sequences for the same user don’t overlap. This table enables checking for missing invoice numbers in each range. The year is an attribute to indicate the period (which can help partition sequences by year or other interval).

CREATE TABLE Invoice (

invoiceID INT PRIMARY KEY,

seqID INT NOT NULL,

invoiceNo INT NOT NULL,

invoiceDate DATE,

totalAmount DECIMAL(12,2),

FOREIGN KEY (seqID) REFERENCES InvoiceSequence(seqID)

-- Optionally, a UNIQUE(seqID, invoiceNo) could be enforced to prevent duplicates

);

Explanation: The Invoice table stores each invoice record. invoiceID is a surrogate primary key. Each invoice is linked to exactly one InvoiceSequence (via seqID foreign key). The combination of seqID and invoiceNo identifies the official invoice number in that sequence. We did not declare UNIQUE(seqID, invoiceNo) here to allow the possibility of duplicate invoice numbers (which normally wouldn’t be allowed, but we simulate an anomaly where a duplicate happened). In a real scenario, we would enforce uniqueness so that a given invoice number cannot be recorded twice for the same sequence/user. invoiceDate is the date on the invoice document, and totalAmount is the total amount of that invoice. This table’s data is used to detect anomalies like duplicate invoices (same invoiceNo appearing twice) and late recording (by comparing with transaction timestamps).

CREATE TABLE TransactionRecord (

transID INT PRIMARY KEY,

invoiceID INT NOT NULL,

accountCategoryID INT NOT NULL,

userID INT NOT NULL,

machineID INT,

amount DECIMAL(12,2),

timestamp DATETIME,

FOREIGN KEY (invoiceID) REFERENCES Invoice(invoiceID),

FOREIGN KEY (accountCategoryID) REFERENCES AccountCategory(categoryID),

FOREIGN KEY (userID) REFERENCES User(userID),

FOREIGN KEY (machineID) REFERENCES Machine(machineID)

);

Explanation: The TransactionRecord table holds individual accounting entries (line items). It has foreign keys to Invoice (the parent invoice it belongs to), AccountCategory (the category of this line), User (the user who input it), and Machine (the device used, if recorded). The userID here should logically match the user who owns the invoice, but we allow it to ensure we can capture if another user somehow entered a transaction (an unlikely scenario, but included for completeness). Each transaction has an amount and a timestamp. By comparing amount sum per invoice to Invoice.totalAmount, we can find mismatches. By looking at timestamp, we analyze patterns (off-hours work, etc.). The presence of machineID allows us to track which machine was used – if null, it means the machine wasn’t recorded (but in our data we generally record it). This table typically would have an index on invoiceID for fast join to Invoice. (In conceptual terms, TransactionRecord is a weak entity of Invoice – its identity is tied to an Invoice.)

CREATE TABLE MachineLog (

logID INT PRIMARY KEY,

machineID INT NOT NULL,

userID INT NOT NULL,

invoiceNo INT,

operationType VARCHAR(20),

timestamp DATETIME,

FOREIGN KEY (machineID) REFERENCES Machine(machineID),

FOREIGN KEY (userID) REFERENCES User(userID)

);

Explanation: The MachineLog table logs machine operations (e.g., scanning an invoice). It has foreign keys to Machine and User – identifying which machine was used by which user. invoiceNo is a reference to the invoice number involved in the operation (not a foreign key to Invoice because the invoice might not exist in the Invoice table if it was never recorded; we store it as a value). For operations like "Scan", we log the invoiceNo scanned. If an invoiceNo in MachineLog has no matching entry in the Invoice table, that indicates a document was scanned but not recorded – our system flags that via the ErrorLog. operationType could be "Scan", "Print", "Email", etc., and timestamp records when it happened. This table allows cross-checking physical document handling with system entries.

CREATE TABLE ErrorLog (

errorID INT PRIMARY KEY,

userID INT NOT NULL,

invoiceID INT,

type VARCHAR(50),

description VARCHAR(255),

timestamp DATETIME,

FOREIGN KEY (userID) REFERENCES User(userID),

FOREIGN KEY (invoiceID) REFERENCES Invoice(invoiceID)

);

Explanation: The ErrorLog table stores anomaly records. userID links to the User responsible for or associated with the error. invoiceID is a foreign key to Invoice if the error involves a specific invoice; it is NULL for errors not tied to a particular invoice (e.g. a user’s pattern issue like "OffHoursPattern"). We allow NULL on invoiceID. The type indicates the category of anomaly (we use values like "MissingInvoice", "DuplicateInvoice", etc.), and description provides details (e.g. "Invoice #1002 was not recorded"). The timestamp is when the error was logged. This table’s entries are generated by analysis processes (or triggers). It allows queries to find, for example, how many errors each user has, or which types of errors are most frequent. The foreign key on invoiceID ensures that if an error references an invoice, that invoice must exist (we would only use it for cases like duplicate or mismatch where the invoice is known; for missing invoices, invoiceID stays NULL because the invoice is not in the system). Views and Stored Programs: To facilitate analysis, we create an example view and a stored function/procedure:

CREATE VIEW ErrorLogView AS

SELECT e.errorID,

u.name AS userName,

COALESCE(i.invoiceNo, NULL) AS invoiceNo,

e.type,

e.description,

e.timestamp

FROM ErrorLog e

LEFT JOIN Invoice i ON e.invoiceID = i.invoiceID

JOIN User u ON e.userID = u.userID;

Explanation: ErrorLogView presents error logs in a more readable format, joining with User to get the user’s name and with Invoice to get the invoice number (if available). We use COALESCE(i.invoiceNo, NULL) so that for errors with no invoice (NULL invoiceID), the view will show NULL for invoiceNo (or we could substitute a label like 'N/A'). This view makes it easy to query anomalies by user name or invoice number directly. For example, one can SELECT \* FROM ErrorLogView WHERE type='MissingInvoice' to list all missing invoice errors with the user names and invoice numbers in question.

CREATE FUNCTION getErrorCount(p\_userID INT)

RETURNS INT

DETERMINISTIC

BEGIN

DECLARE errCount INT;

SELECT COUNT(\*) INTO errCount

FROM ErrorLog

WHERE userID = p\_userID;

RETURN errCount;

END;

Explanation: getErrorCount is a user-defined function that returns the number of errors recorded for a given user (identified by p\_userID). It simply counts ErrorLog entries for that user. We declare it DETERMINISTIC (given same input, output doesn’t change unless data changes) and it returns an INT. This function can be used in queries – for instance, to list all users with their error counts, one could do: SELECT name, getErrorCount(userID) AS errorCount FROM User;. This encapsulates the logic of counting errors per user.

SHOW CREATE TABLE ErrorLog;

ALTER TABLE ErrorLog MODIFY errorID INT NOT NULL AUTO\_INCREMENT;

DELIMITER $$

DROP PROCEDURE IF EXISTS CheckMissingInvoices$$

CREATE PROCEDURE CheckMissingInvoices()

BEGIN

/\* digits 0‑9 as an inline derived table \*/

INSERT INTO ErrorLog (userID, invoiceID, type, description, timestamp)

SELECT seq.userID,

NULL,

'MissingInvoice',

CONCAT('Invoice #', seq.startNo + d.n,

' was missing (not recorded by userID ',

seq.userID, ').'),

NOW()

FROM InvoiceSequence AS seq

JOIN ( SELECT 0 AS n UNION ALL SELECT 1 UNION ALL SELECT 2 UNION ALL

SELECT 3 UNION ALL SELECT 4 UNION ALL SELECT 5 UNION ALL

SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9

) AS d

ON seq.startNo + d.n <= seq.endNo -- keep within the range

LEFT JOIN Invoice i

ON i.seqID = seq.seqID

AND i.invoiceNo = seq.startNo + d.n

WHERE i.invoiceID IS NULL; -- only truly missing numbers

END$$

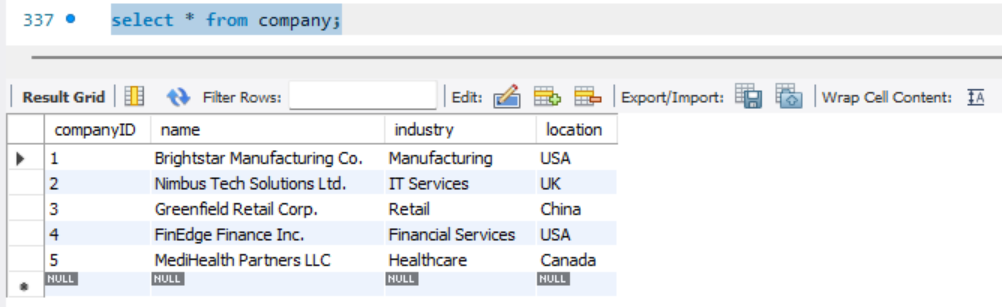
DELIMITER ;

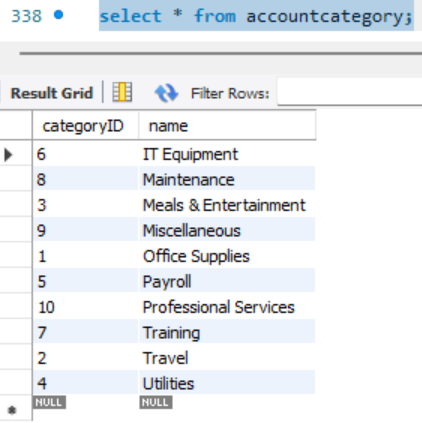
Explanation:  
For every invoice number that should exist in each user’s 10‑number InvoiceSequence but doesn’t, the procedure inserts a “MissingInvoice” row into ErrorLog—all in a single set‑based statement, no loops or cursors.

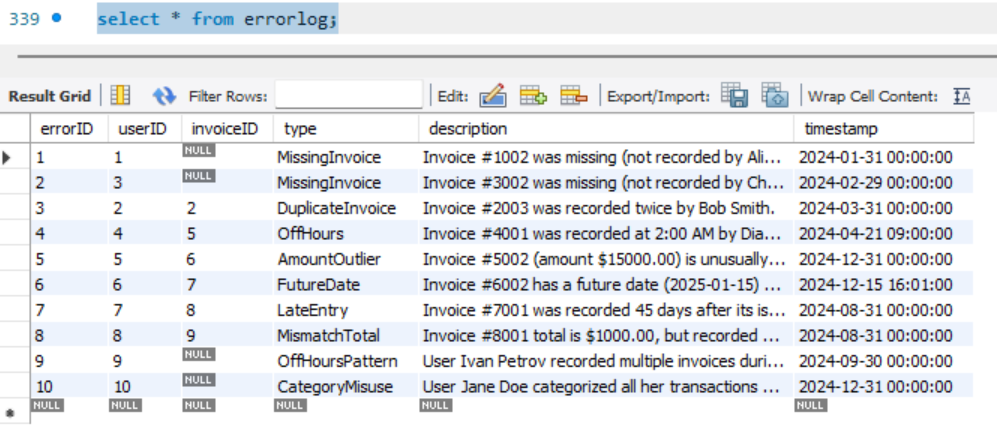
Run CALL CheckMissingInvoices(); whenever you want to refresh the log of missing invoices.

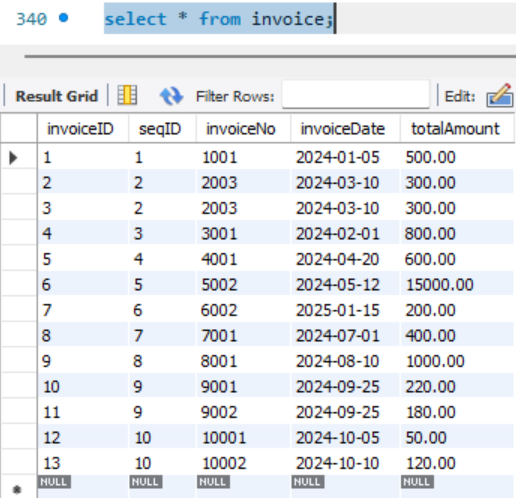
# Print Results

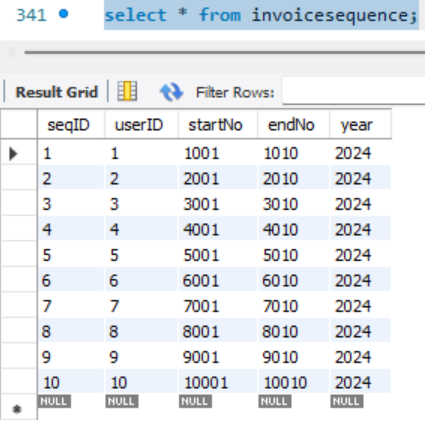
Table results print screenshot:

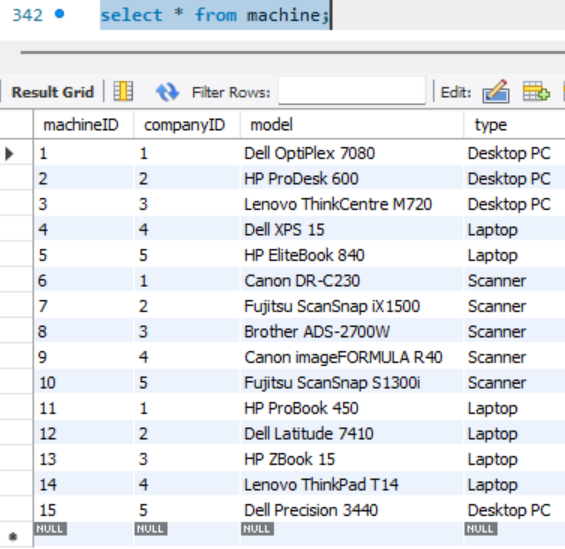


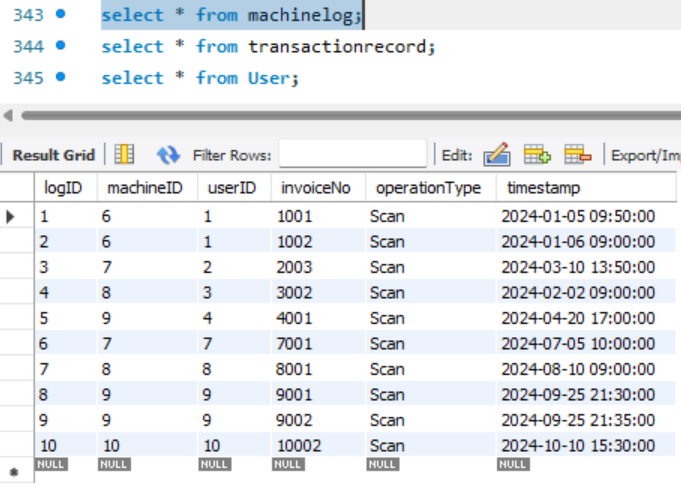


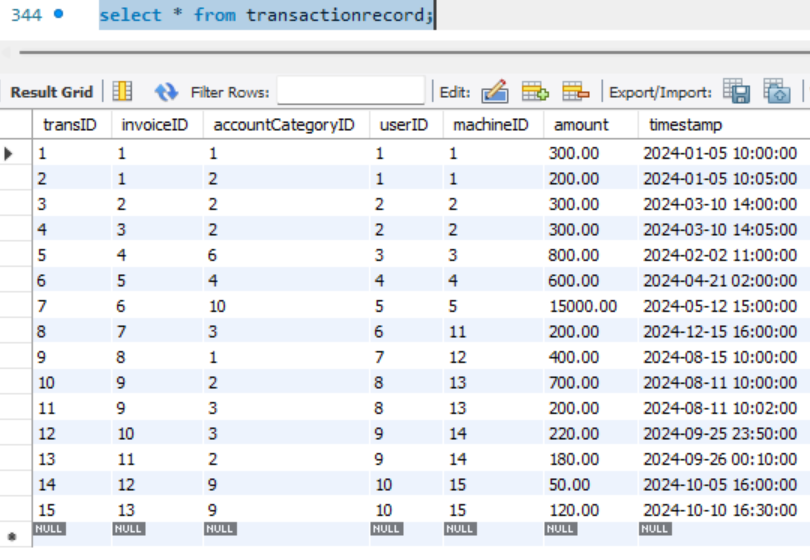


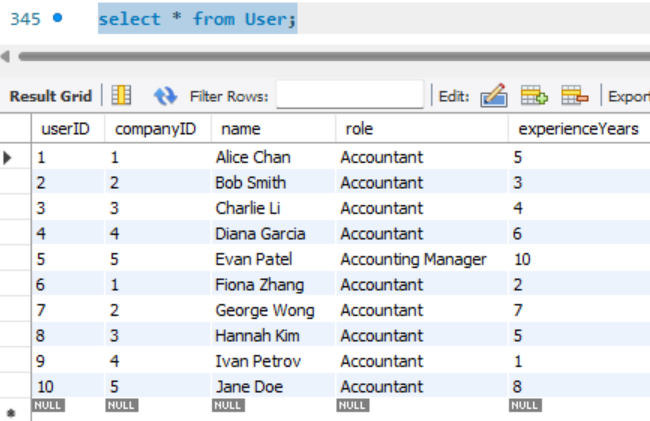


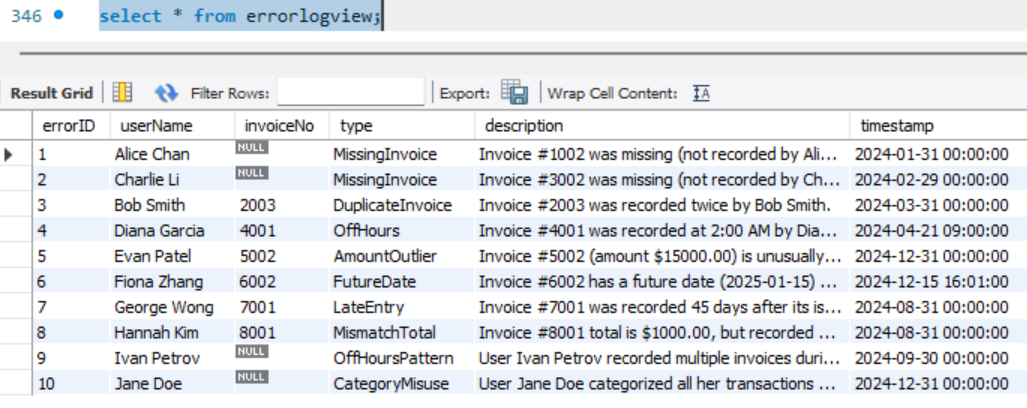


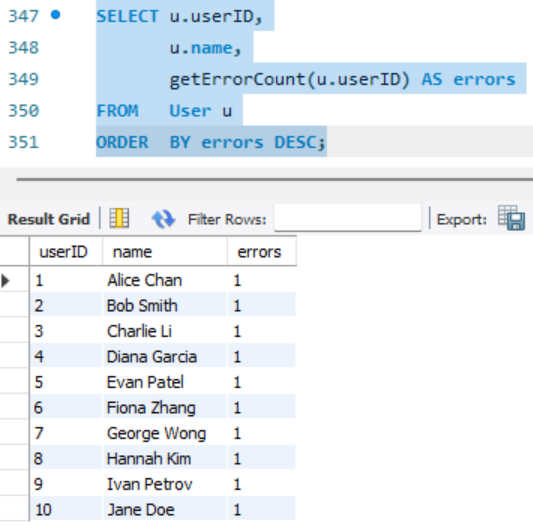


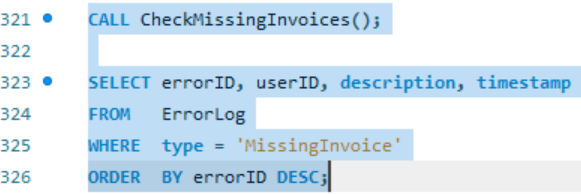


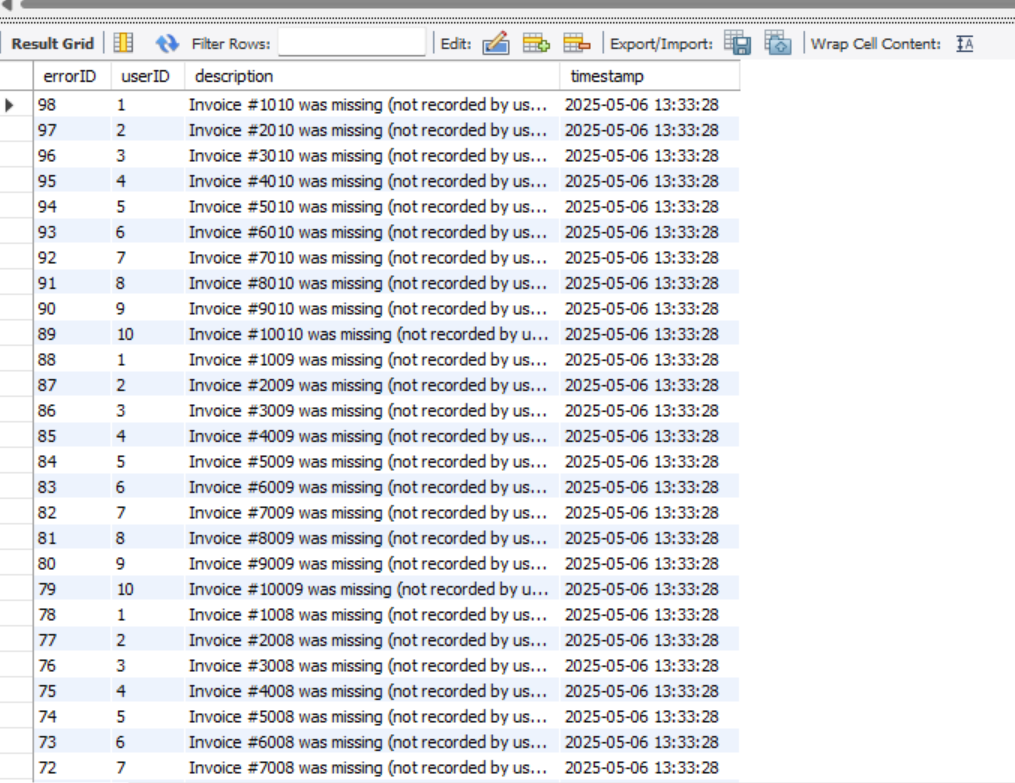






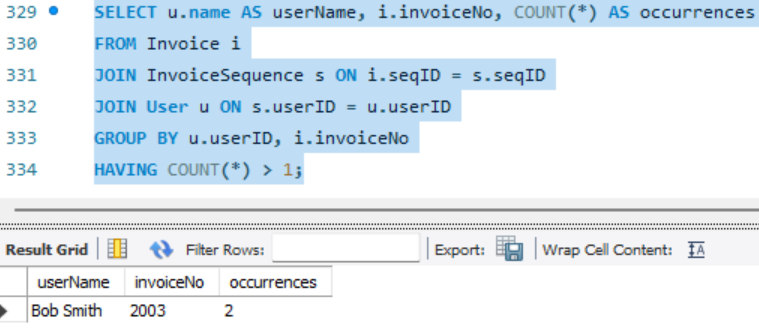




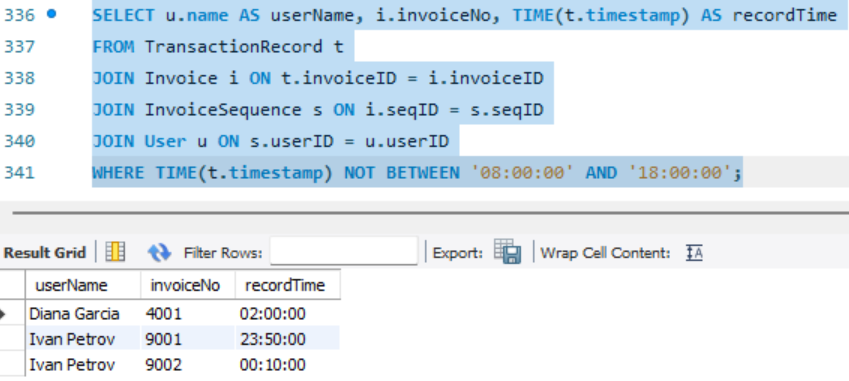


# SQL Query

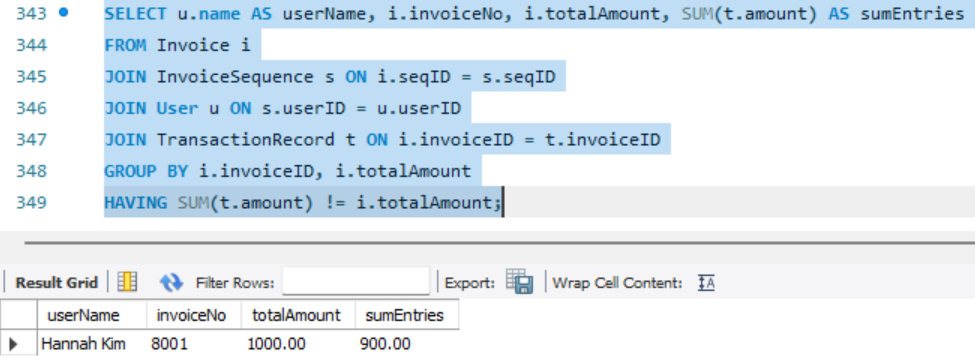
Query 1: Find any invoice numbers that were recorded more than once (duplicate invoices) and by whom.



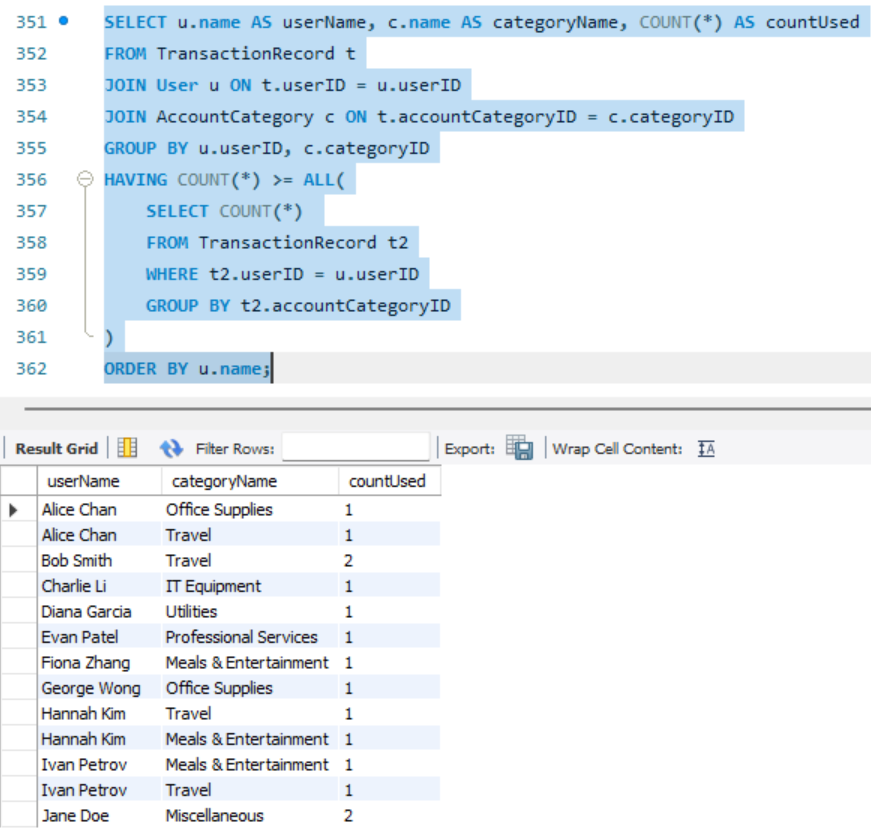
Query 2: List all transactions that were recorded outside of normal business hours (assume 8:00–18:00 as normal).



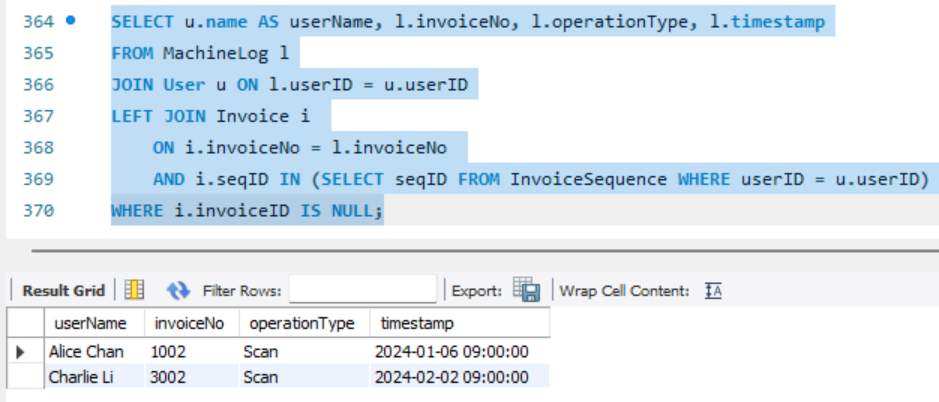
Query 3: Check for any invoice where the total amount does not equal the sum of its transaction amounts.



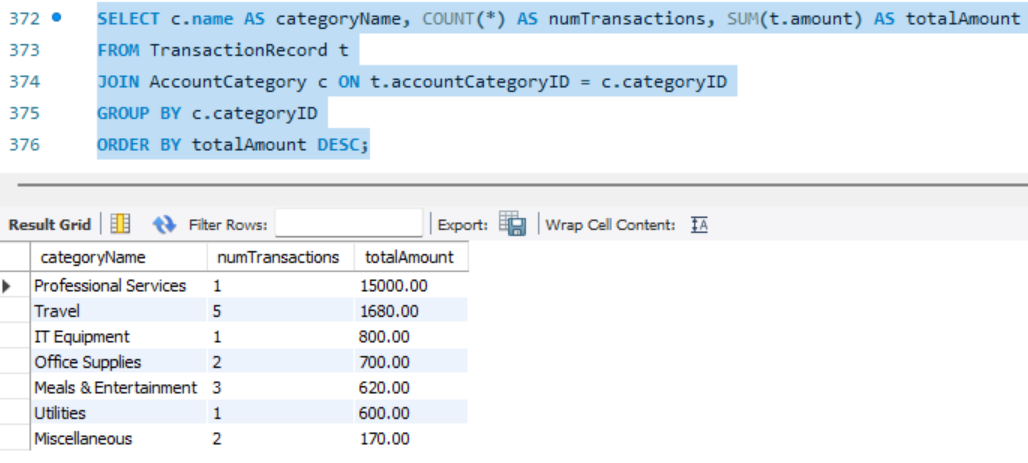
Query 4: For each user, find their most frequently used account category in transactions.



Query 5: List any invoices that were scanned by a user but not recorded in the system (i.e. appear in MachineLog but not in Invoice).



Query 6: Show the total number of transactions and total amount for each account category (ordered by total amount descending). This aggregates TransactionRecord continuing from Query 5.



# Strong points

* Our system doesn’t just store your accounting data—it’s like a watchful assistant that keeps an eye out for mistakes. Picture a librarian who not only organizes books but also notices when one is missing. That’s what we’ve built here. It spots things like missing invoices, duplicates, or entries made way too late. For example, it checks the sequence of your invoice numbers and flags any gaps, hinting that an invoice might have gone AWOL. This isn’t just about catching errors—it’s about learning from them. Every time it finds a slip-up, it logs it with details so you can see what happened. Over time, you can spot patterns (like “Oops, we keep missing invoices on Fridays!”) and help your team get better at bookkeeping. It’s like having a coach built right into the system.
* We’ve done something pretty cool: we’ve linked the machines you use—like computers and scanners—to your financial records. Think of it as a detective on your team, piecing together clues. It can tell if an invoice was scanned but never entered into the system, or if someone’s burning the midnight oil on a specific computer. Most accounting systems don’t pay attention to this stuff, but we do. This connection lets you catch mistakes—like that missing invoice that got scanned but forgotten—and even figure out how your team works. Maybe one computer is always slow, or someone’s habits are causing delays. It’s a smarter way to keep everything in sync and running smoothly.