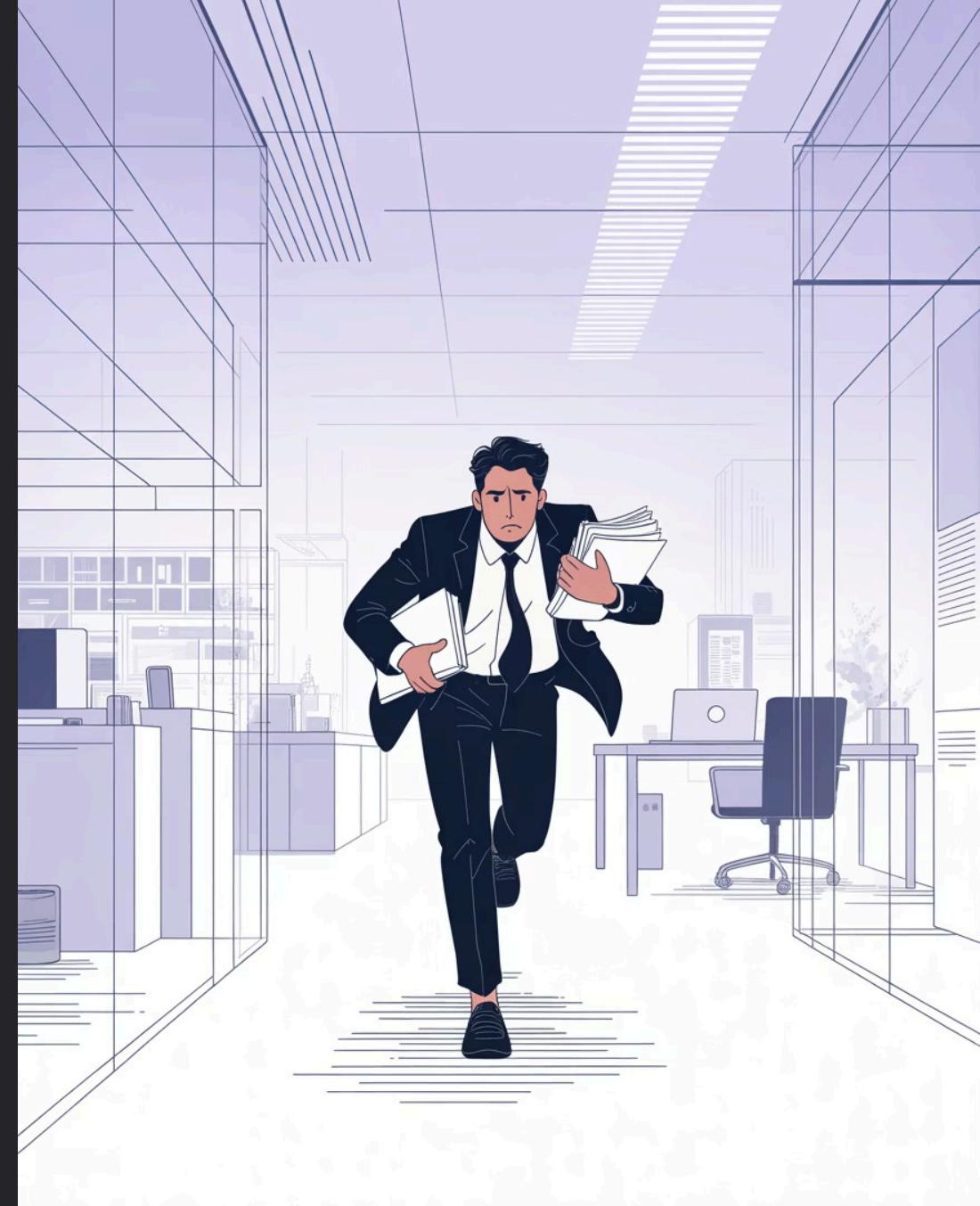


Temp Tables & Table Variables

Complex Multi-Step Workflows

NAD Banking Month-End Reconciliation

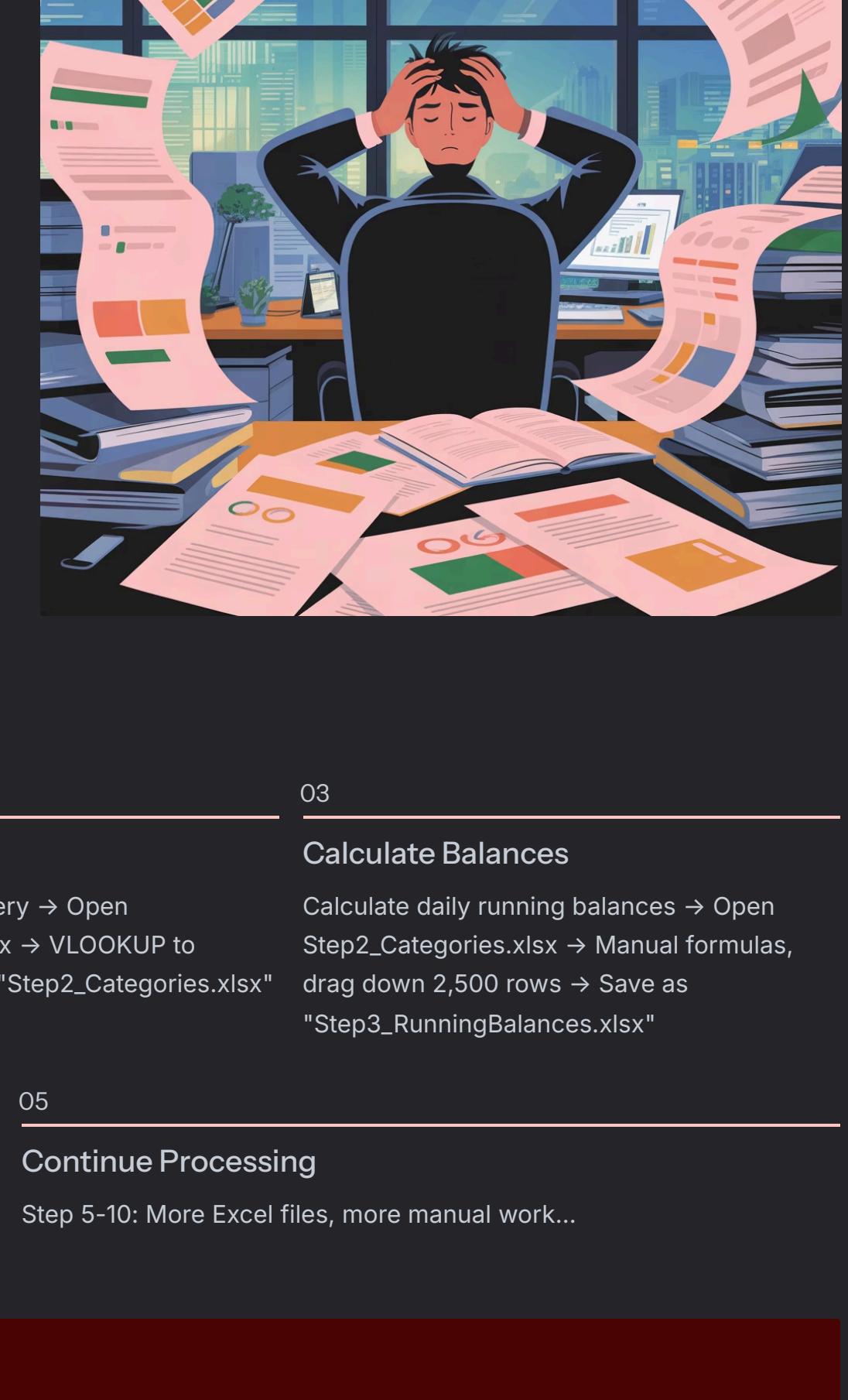
Video 10 of 15 | Duration: 17 minutes | Level: Advanced



5:00 PM Friday - Finance Manager Walks In (Stressed)

"Month-end reconciliation is due Monday morning for the board meeting!"

"I need the full December reconciliation report... but it takes **3 HOURS** to complete manually!"



The 10-Step Manual Nightmare

01	02	03
Extract Transactions Run query to extract all December transactions → Copy 2,500+ rows to Excel → Save as "Step1_Transactions.xlsx"	Categorise Run categorisation query → Open Step1_Transactions.xlsx → VLOOKUP to categorise → Save as "Step2_Categories.xlsx"	Calculate Balances Calculate daily running balances → Open Step2_Categories.xlsx → Manual formulas, drag down 2,500 rows → Save as "Step3_RunningBalances.xlsx"

04	05
Identify Overdrafts Identify overdraft accounts → Filter Step3 for negative balances → Save as "Step4_Overdrafts.xlsx"	Continue Processing Step 5-10: More Excel files, more manual work...

Total: 10 Excel files, 3 hours of copy-paste hell!

The Problem - Sequential Dependencies

- Can't do Step 5 without Step 3 results
- Can't do Step 8 without Steps 6 & 7 complete
- Each step DEPENDS on previous steps!

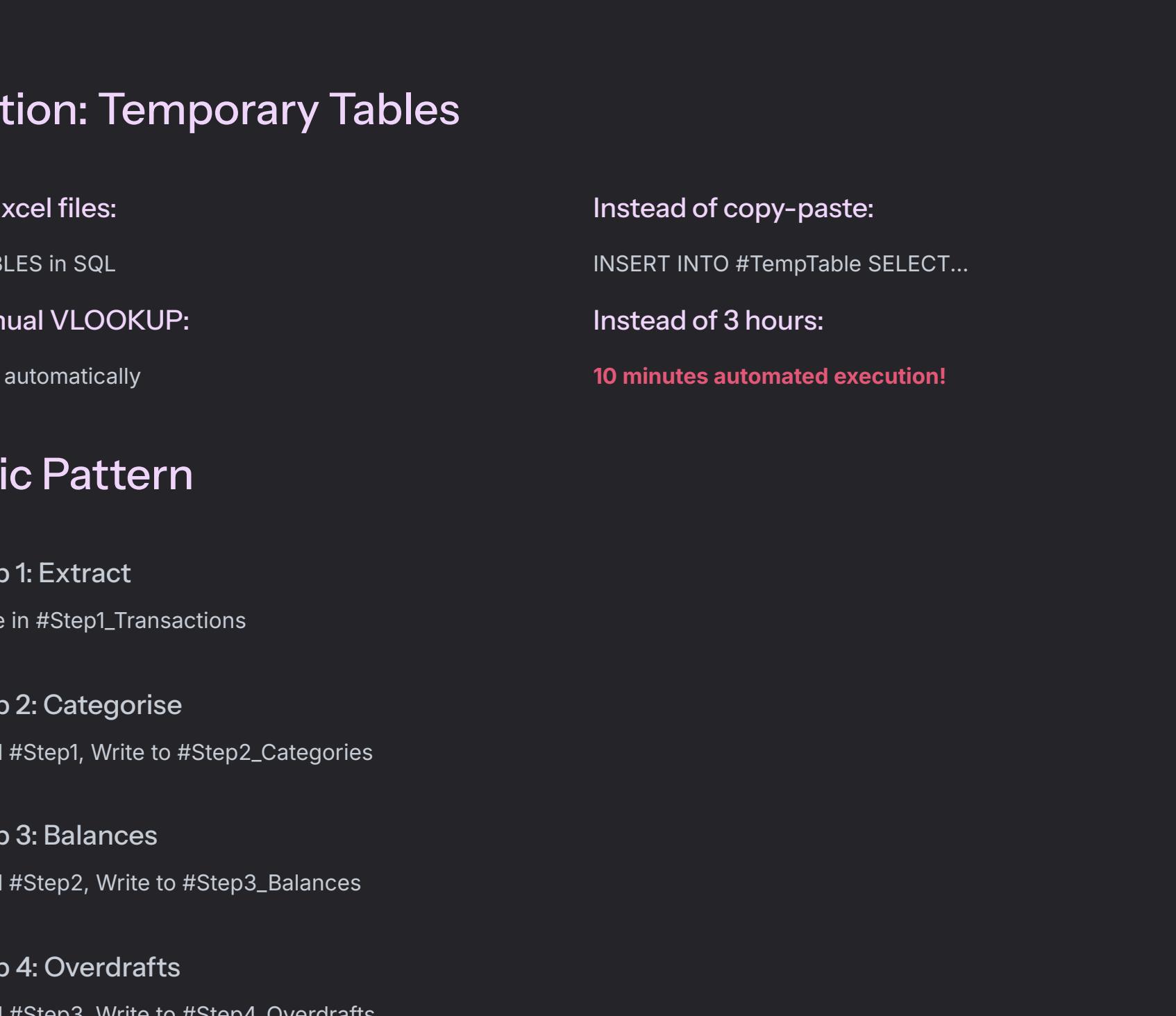
Traditional SQL Problem:

Run query 1 → Can't save results for query 2
Run query 2 → Loses query 1 results
Can't chain queries together!

Solution needed:

- Store intermediate results
- Each step reads previous step's data
- Automated chain of processing

The Copy-Paste Disaster



Errors discovered:

- Wrong VLOOKUP range (Step 2)
- Formulas not copied to all rows (Step 3)
- Manual filter missed rows (Step 4)

Weekend ruined fixing errors! 😞

What If We Could...

Run ONE script?

Store intermediate results automatically?

Chain all 10 steps together?

Complete in 10 minutes instead of 3 hours?

That's what TEMP TABLES do!

The Solution: Temporary Tables

Instead of 10 Excel files:

Create TEMP TABLES in SQL

Instead of manual VLOOKUP:

JOIN temp tables automatically

Instead of copy-paste:

INSERT INTO #TempTable SELECT...

Instead of 3 hours:

10 minutes automated execution!

Total:

45 minutes to automate (including testing)

Future: 10 minutes each month forever!

Monday 8 AM:

Finance Manager presents perfect report to board

You: Weekend saved! 🎉

PLUS: 5 Practice Exercises with Solutions

The Friday Evening Victory

5:00 PM
"I need month-end reconciliation by Monday!"

5:30 PM
Temp tables script written

6:00 PM
Tested with November data

6:30 PM
Deployed for December

EXEC MonthEndReconciliation @Year = 2024, @Month = 12

6:45 PM
Complete! All 10 steps done!

Total:

45 minutes to automate (including testing)

Future: 10 minutes each month forever!

Monday 8 AM:

Finance Manager presents perfect report to board

You: Weekend saved! 🎉

PLUS: 5 Practice Exercises with Solutions

The Friday Evening Victory

5:00 PM
"I need month-end reconciliation by Monday!"

5:30 PM
Temp tables script written

6:00 PM
Tested with November data

6:30 PM
Deployed for December

EXEC MonthEndReconciliation @Year = 2024, @Month = 12

6:45 PM
Complete! All 10 steps done!

Total:

45 minutes to automate (including testing)

Future: 10 minutes each month forever!

Monday 8 AM:

Finance Manager presents perfect report to board

You: Weekend saved! 🎉

TEMP TABLES - THE WORKHORSE

Session-Spaced Intermediate Storage

What Are Temp Tables?

Definition:

- Temporary tables stored in tempdb database
- Exist only for your session (connection)
- Automatically dropped when session ends
- Prefix with # (single hash mark)

Think of them as:

- Scratch paper for complex calculations
- Staging area for multi-step processes
- Intermediate results holder

Lifetime:

- Created:** When you run CREATE TABLE #Name
- Exists:** Throughout your entire session
- Dropped:** When session disconnects OR you DROP manually

Creating a Temp Table

Syntax:

```
CREATE TABLE #TempTableName (  
    column1 datatype,  
    column2 datatype,  
    ...  
)
```

Real NAB Example:

```
CREATE TABLE #MonthlyTransactions (  
    account_id INT,  
    account_number NVARCHAR(20),  
    customer_name NVARCHAR(100),  
    transaction_count INT,  
    total_deposits DECIMAL(15,2),  
    total_withdrawals DECIMAL(15,2),  
    net_change DECIMAL(15,2)  
)
```

Looks like regular table! Just has # prefix

Populating Temp Tables

Method 1: INSERT INTO ... SELECT

```
INSERT INTO #MonthlyTransactions  
SELECT  
    a.account_id,  
    a.account_number,  
    a.customer_name,  
    COUNT(*) AS transaction_count,  
    SUM(CASE WHEN amount > 0 THEN amount ELSE 0 END) AS total_deposits,  
    SUM(CASE WHEN amount < 0 THEN ABS(amount) ELSE 0 END) AS total_withdrawals,  
    SUM(amount) AS net_change  
FROM accounts a  
LEFT JOIN transactions t ON a.account_id = t.account_id  
WHERE t.transaction_date >= '2024-12-01'  
GROUP BY a.account_id, a.account_number, a.customer_name;
```

Standard INSERT syntax!

Method 2: SELECT INTO (shorthand)

```
SELECT  
    account_id,  
    SUM(amount) AS total  
INTO #QuickTemp  
FROM transactions  
GROUP BY account_id;
```

Creates temp table automatically!

Column names and types inferred!

Querying Temp Tables

Just like regular tables:

```
SELECT *  
FROM #MonthlyTransactions  
WHERE net_change > 0  
ORDER BY transaction_count DESC;
```

Can use in JOINS:

```
SELECT  
    mt.customer_name,  
    mt.net_change,  
    a.account_type  
FROM #MonthlyTransactions mt  
JOIN accounts a  
    ON mt.account_id = a.account_id;
```

Full SQL power available!

Where Are Temp Tables Stored?

Physically:

tempdb database

Check with:

```
SELECT name, create_date  
FROM tempdb.sys.tables  
WHERE name LIKE '#MonthlyTransactions%';
```

SQL Server adds unique suffix:

#MonthlyTransactions__000000000123

Why?

Prevent naming conflicts between sessions

Session Scope - What Does It Mean?

YOUR session ONLY:

- You create #MyTemp
- You can see #MyTemp
- You can query #MyTemp

OTHER sessions:

- They create #MyTemp (same name!)
- They see THEIR #MyTemp (different table!)
- They CANNOT see YOUR #MyTemp

Each session has isolated temp tables!

Cleanup - Two Ways

Automatic (recommended):

Session ends → All temp tables automatically dropped

No action needed!

Manual (when needed):

```
DROP TABLE #MonthlyTransactions;
```

Use manual DROP when:

- Reusing same session for multiple tasks
- Large temp table (free up tempdb space)
- Testing/development (clean slate between runs)

Temp Table Characteristics

Can add indexes

```
CREATE INDEX IX_Account ON #MonthlyTransactions(account_id);
```

Can add constraints

```
ALTER TABLE #Temp ADD CONSTRAINT PK_ID PRIMARY KEY (id);
```

Has statistics (query optimiser uses them)

```
UPDATE STATISTICS #MonthlyTransactions;
```

Supports transactions

```
BEGIN TRANSACTION;  
INSERT INTO #Temp...;  
COMMIT;
```

Can participate in JOIN, WHERE, ORDER BY

Just like permanent tables!

When to Use Temp Tables

Use temp tables when:

- Large datasets (1,000+ rows)
- Reused multiple times in script
- Multi-step workflows (step results)
- Need indexes for performance
- Complex transformations
- Data persists across batches (GO statements)

Examples:

- Month-end reconciliation (step-by-step processing)
- Complex ETL workflows (extract → transform → load)
- Report generation (aggregate → filter → format)
- Data quality checks (identify → categorise → report)

When NOT to Use Temp Tables

Don't use temp tables when:

- Small dataset (< 100 rows) → Use table variable
- Single query only → Use CTE instead
- Simple aggregation → Use subquery
- Immediate one-time result → Just SELECT

Overkill examples:

- Quick COUNT(*) check
- Simple INNER JOIN query
- One-time data exploration
- Ad-hoc stakeholder question

Batch-Scooped Quick Storage

What Are Table Variables?



Definition:

- In-memory table structure
- Declared with DECLARE (not CREATE)
- Batch-scoped (shorter lifetime than temp tables)
- Prefix with @ (single at-sign)



Think of them as:

- Variables that hold table data
- Quick staging for small datasets
- Lightweight temp table alternative



Lifetime:

- **Declared:** In current batch only
- **Exists:** Until end of batch (GO statement)
- **Dropped:** Automatically at batch end

Declaring a Table Variable

Syntax:

```
DECLARE @VariableName TABLE (
    column1 datatype,
    column2 datatype,
    ...
);
```

Real NAB Example:

```
DECLARE @AccountSummary TABLE (
    account_id INT,
    customer_name NVARCHAR(100),
    current_balance DECIMAL(15,2),
    risk_level NVARCHAR(20)
);
```

Looks like temp table, but DECLARE instead of CREATE!

Populating Table Variables

INSERT INTO @VariableName

```
INSERT INTO @AccountSummary
SELECT
    a.account_id,
    a.customer_name,
    a.opening_balance + ISNULL(SUM(t.amount), 0) AS current_balance,
    CASE
        WHEN a.opening_balance + ISNULL(SUM(t.amount), 0) < 1000 THEN 'High Risk'
        WHEN a.opening_balance + ISNULL(SUM(t.amount), 0) < 5000 THEN 'Medium Risk'
        ELSE 'Low Risk'
    END AS risk_level
FROM accounts a
LEFT JOIN transactions t ON a.account_id = t.account_id
GROUP BY a.account_id, a.customer_name, a.opening_balance;
```

Same INSERT syntax as temp tables!

Querying Table Variables

```
SELECT *
FROM @AccountSummary
WHERE risk_level = 'High Risk'
ORDER BY current_balance;
```

Can JOIN with other tables:

```
SELECT
    a.account_type,
    av.risk_level,
    COUNT(*) AS account_count
FROM @AccountSummary av
JOIN accounts a
    ON av.account_id = a.account_id
GROUP BY a.account_type, av.risk_level;
```

Full query capabilities!

Key Difference - Batch Scope

Batch = Code between GO statements

Example - Table Variable Dies at GO:

```
DECLARE @Temp TABLE (id INT);
INSERT INTO @Temp VALUES (1);
SELECT * FROM @Temp; --
```

The Complete Comparison & Decision Guide

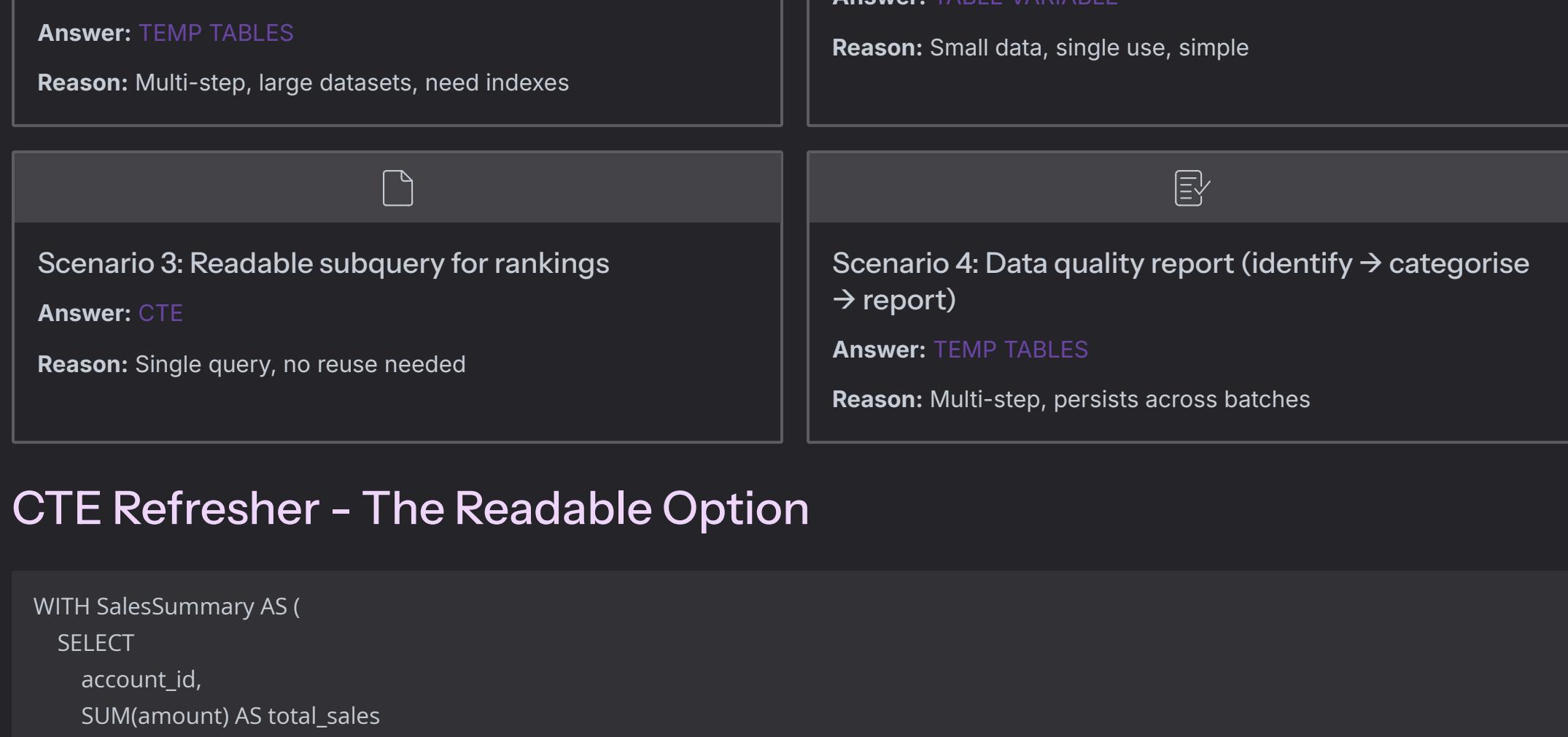
The Three Options

Option 1: Temp Tables (#TempTable)	Option 2: Table Variables (@TableVariable)	Option 3: CTEs (Common Table Expressions)
<ul style="list-style-type: none"> Physical storage in tempdb Session-scoped Can add indexes Has statistics Best for large datasets & multi-step workflows 	<ul style="list-style-type: none"> In-memory structure (actually in tempdb too!) Batch-scoped Limited indexing No statistics Best for small datasets & simple staging 	<ul style="list-style-type: none"> Query-scoped No storage (inline with query) Calculated on demand Readable syntax Best for single-query readability

Feature Comparison Table

Feature	Temp Table	Table Variable	CTE
Storage location	tempdb	tempdb	None
Lifetime	Session	Batch	Query
Add indexes after create	✓ Yes	✗ No	N/A
Statistics	✓ Yes	✗ No	✗ No
Reuse across batches	✓ Yes	✗ No	✗ No
Transactions	✓ Yes	✓ Yes	✓ Yes
Explicit DROP needed	Optional	No	No
Ideal dataset size	> 1,000	< 1,000	Any
Performance (large data)	Excellent	Poor	Good

When to Use Each - Decision Tree



Real-World Scenarios

<p>Scenario 1: Month-end reconciliation (10 steps, large data)</p> <p>Answer: TEMP TABLES</p> <p>Reason: Multi-step, large datasets, need indexes</p>	<p>Scenario 2: Quick parameter list for IN clause (20 IDs)</p> <p>Answer: TABLE VARIABLE</p> <p>Reason: Small data, single use, simple</p>
---	--

<p>Scenario 3: Readable subquery for rankings</p> <p>Answer: CTE</p> <p>Reason: Single query, no reuse needed</p>	<p>Scenario 4: Data quality report (identify → categorise → report)</p> <p>Answer: TEMP TABLES</p> <p>Reason: Multi-step, persists across batches</p>
---	---

CTE Refresher - The Readable Option

```
WITH SalesSummary AS (
    SELECT
        account_id,
        SUM(amount) AS total_sales
    FROM transactions
    WHERE transaction_date >= '2024-12-01'
    GROUP BY account_id
)
SELECT
    a.customer_name,
    s.total_sales
FROM accounts a
JOIN SalesSummary s ON a.account_id = s.account_id
WHERE s.total_sales > 10000;
```

CTE = Inline, readable, single query

Perfect for organising complex SELECT statements!

When CTE is NOT enough:

- ✗ Need to query the CTE multiple times (recalculated each time!)
- ✗ Need to store for later processing
- ✗ Dataset too large for inline processing

Temp Tables vs CTEs - The Reuse Test

With CTE (BAD - recalculates 3 times):

```
WITH Summary AS (SELECT ... complex calculation ...)
SELECT * FROM Summary WHERE condition1; -- Calculate once
SELECT * FROM Summary WHERE condition2; -- Calculate again!
SELECT * FROM Summary WHERE condition3; -- Calculate third time!
```

Inefficient! Complex query runs 3 times!

Rule: Reuse results → Temp table, not CTE!

Global Temp Tables (##GlobalTemp)

Special case: Cross-session sharing

```
CREATE TABLE ##SharedData (
    id INT,
    value NVARCHAR(100)
);
```

Visible to ALL sessions! Prefix with ## (double hash)

Use case:

- Session A creates ##Lookup table
- Sessions B, C, D all read ##Lookup
- Shared reference data across team

Caution:

- Dropped when LAST session using it disconnects
- Naming conflicts possible (other users may create same name)
- Use sparingly!

Best practice: Permanent table better for true shared data

Performance Tips

Tip 1: Index temp tables for large datasets

```
CREATE INDEX IX_AccountID ON #Transactions(account_id);
```

Massive performance boost for JOINs and WHERE!

Tip 2: Use table variables for < 1,000 rows

Less overhead, faster for small data

Tip 3: Use CTEs for readable one-time queries

No storage overhead, clean syntax

Tip 4: DROP temp tables when done (in long scripts)

Frees up tempdb space immediately

Tip 5: Monitor tempdb usage

Large temp tables can fill tempdb

Watch for "tempdb is full" errors

MULTI-STEP WORKFLOWS - THE PRODUCTION PATTERN

Chaining Temp Tables for Complex Processing

The Month-End Reconciliation Challenge

01	Extract 15,000 December transactions	02	Calculate daily running balances per account
03	Identify overdraft instances	04	Flag suspicious large transactions
05	Calculate fee revenue	06	Match deposits to withdrawals
07	Identify unmatched transactions	08	Calculate interest accruals
09	Generate variance report	10	Final reconciliation summary

Each step feeds the NEXT step!

The Temp Table Chain Pattern

```
Step 1 → #Step1_Transactions
↓
Step 2 reads #Step1, writes #Step2_DailyBalances
↓
Step 3 reads #Step2, writes #Step3_Overdrafts
↓
Step 4 reads #Step3, writes #Step4_SuspiciousTransactions
↓
Step 5 reads #Step1 + #Step2, writes #Step5_FeeRevenue
↓
... continues through Step 10
↓
Final: Read ALL temp tables, generate report!
```

Automated pipeline!

Code Structure - The Template

```
-- STEP 1: Extract base data
CREATE TABLE #Step1_Transactions (...);
INSERT INTO #Step1_Transactions SELECT ... FROM source;
CREATE INDEX IX_Account ON #Step1_Transactions(account_id);

-- STEP 2: Transform/Calculate
CREATE TABLE #Step2_Calculated (...);
INSERT INTO #Step2_Calculated SELECT ... FROM #Step1_Transactions;
CREATE INDEX IX_Date ON #Step2_Calculated(date_column);

-- STEP 3: Further processing
CREATE TABLE #Step3_Results (...);
INSERT INTO #Step3_Results SELECT ... FROM #Step2_Calculated;

-- FINAL: Generate report
SELECT ...
FROM #Step1_Transactions t1
JOIN #Step2_Calculated t2 ON ...
JOIN #Step3_Results t3 ON ...

-- CLEANUP
DROP TABLE #Step1_Transactions;
DROP TABLE #Step2_Calculated;
DROP TABLE #Step3_Results;
```

Clear, organised, maintainable!

Real NAB Example - Step by Step

STEP 1: Extract Transactions

```
CREATE TABLE #Step1_Transactions (
    transaction_id INT,
    account_id INT,
    transaction_date DATETIME,
    amount DECIMAL(15,2)
);

INSERT INTO #Step1_Transactions
SELECT
    transaction_id,
    account_id,
    transaction_date,
    amount
FROM transactions
WHERE transaction_date >= '2024-12-01'
    AND transaction_date < '2025-01-01';

CREATE INDEX IX_Account ON #Step1_Transactions(account_id);
```

Result: 15,248 December transactions stored

STEP 2: Daily Running Balances

```
CREATE TABLE #Step2_DailyBalances (
    account_id INT,
    balance_date DATE,
    running_balance DECIMAL(15,2)
);

INSERT INTO #Step2_DailyBalances
SELECT
    t.account_id,
    CAST(t.transaction_date AS DATE),
    a.opening_balance + SUM(t.amount) OVER (
        PARTITION BY t.account_id
        ORDER BY CAST(t.transaction_date AS DATE)
    ) AS running_balance
FROM #Step1_Transactions t
JOIN accounts a ON t.account_id = a.account_id
GROUP BY t.account_id, CAST(t.transaction_date AS DATE), a.opening_balance;
```

Result: Daily balances calculated using previous step's data!

Uses STEP 2 results!

STEP 3: Identify Overdrafts

```
CREATE TABLE #Step3_Overdrafts (
    account_id INT,
    overdraft_date DATE,
    overdraft_amount DECIMAL(15,2)
);

INSERT INTO #Step3_Overdrafts
SELECT
    account_id,
    balance_date,
    running_balance
FROM #Step2_DailyBalances
WHERE running_balance < 0;
```

Result: 23 overdraft instances identified!

Uses STEP 2 results!

STEP 4: Flag Suspicious Transactions

```
CREATE TABLE #Step4_Suspicious (
    transaction_id INT,
    amount DECIMAL(15,2),
    flag_reason NVARCHAR(100)
);

INSERT INTO #Step4_Suspicious
SELECT
    transaction_id,
    amount,
    CASE
        WHEN ABS(amount) > 10000 THEN 'Large transaction (>$10k)'
        WHEN ABS(amount) > 5000 THEN 'Medium alert (>$5k)'
    END
FROM #Step1_Transactions
WHERE ABS(amount) > 5000;
```

Result: 47 suspicious transactions flagged!

Uses STEP 1 results!

Final Report - Combining All Steps

```
SELECT
    a.account_number,
    a.customer_name,
    a.opening_balance,
    SUM(t.amount) AS total_movement,
    a.opening_balance + SUM(t.amount) AS closing_balance,
    COUNT(DISTINCT o.overdraft_date) AS overdraft_days,
    COUNT(DISTINCT s.transaction_id) AS suspicious_count
FROM MonthEndReconciliation a
LEFT JOIN #Step1_Transactions t ON a.account_id = t.account_id
LEFT JOIN #Step2_DailyBalances o ON a.account_id = o.account_id
LEFT JOIN #Step4_Suspicious s ON a.account_id = s.account_id
GROUP BY a.account_number, a.customer_name, a.opening_balance
ORDER BY closing_balance DESC;
```

All solutions in SQL file! Try first, then check answers.

WHAT YOU MASTERED

Technical Skills:

- ✓ Temp table creation and management (#TempTable)
- ✓ Table variable declaration (@TableVarible)
- ✓ Global temp tables (#GlobalTemp)
- ✓ Session Variables (query scope)
- ✓ Indexing temp tables for performance
- ✓ Multi-step workflow patterns
- ✓ When to use each storage type
- ✓ Production procedure patterns

Business Skills:

- ✓ Month-end reconciliation automation
- ✓ Complex multi-step processing
- ✓ Financial reporting workflows
- ✓ Audit trial creation
- ✓ Intermediate results storage

Career Skills:

- ✓ Enterprise-grade workflow design
- ✓ Performance optimisation techniques
- ✓ Production-ready code structure
- ✓ Debugging multi-step processes
- ✓ Code organisation best practices
- ✓ Interview-ready for complex SQL scenarios

Real-World Applications

Retail:

- Inventory reconciliation
- Sales aggregation pipelines
- Customer segmentation workflows
- Promotion effectiveness analysis

Healthcare:

- Patient billing reconciliation
- Claim processing workflows
- Multi-step diagnosis validation
- Treatment outcome tracking

Manufacturing:

- Production quality checks
- Multi-stage inspection reports
- Defect tracking workflows
- Equipment maintenance schedules

Banking (NAB example):

- Month-end reconciliation needed
- 5:00 PM: Manual Excel nightmare (10 files, 3.5 hours)
- Errors: VLOOKUP mistakes, formula errors, copy-paste failures
- Stress level: Maximum 😱
- Weekends: Ruined fixing errors

Banking (NAB example):

- 5:30 PM: One-time script development (temp table chain)
- 6:00 PM: Testing complete
- 6:30 PM: EXEC MonthEndReconciliation @Year = 2024, @Month = 12
- 6:40 PM: Complete! Perfect! 🎉

Banking (NAB example):

- Error rate: 0%
- Stress level: Zero 😊
- Weekends: Free!

The Friday Evening Miracle

BEFORE Temp Tables:

- 5:00 PM: Month-end reconciliation needed
- 5:00-8:30 PM: Manual Excel nightmare (10 files, 3.5 hours)
- Errors: VLOOKUP mistakes, formula errors, copy-paste failures
- Stress level: Maximum 😱
- Weekends: Ruined fixing errors

AFTER Temp Tables:

- 5:30 PM: Write script with temp tables (one time!)
- 6:00 PM: Test with sample data
- 6:30 PM: EXEC MonthEndReconciliation @Year = 2024, @Month = 12
- 6:40 PM: Complete! Perfect! 🎉

Banking (NAB example):

- Future months: 10 minutes forever! 💡

Banking (NAB example):

Decision Guide Summary

Question: What should I use?

< 100 rows, single use → CTE (inline query)	< 1,000 rows, single batch → TABLE VARIABLE (@Variable)
> 1,000 rows OR multi-step → TEMP TABLE (#TempTable)	Cross-session sharing → GLOBAL TEMP TABLE (#GlobalTemp)

Performance:

Need indexes? → TEMP TABLE

Scope:

Need across batches? → TEMP TABLE

Simplicity:

Quick staging? → TABLE VARIABLE

Readability:

Single query? → CTE

Best Practices Checklist

- ✓ Use temp tables for large datasets (>1,000 rows)
- ✓ Add indexes to temp tables for performance
- ✓ Use table variables for small datasets (<1,000 rows)
- ✓ Use CTEs for single-query readability
- ✓ DROP temp tables when done (in long scripts)

- ✓ Month-end reconciliation automation
- ✓ Complex multi-step processing
- ✓ Financial reporting workflows
- ✓ Audit trial creation
- ✓ Intermediate results storage

Interview Value

Simplicity:

"How would you handle complex multi-step processing?"

→ Temp tables! 🎯

Business scenario:

"What's the difference between temp table and table variable?"

"When would you use a CTE vs temp table?"

→ Single use vs reuse! 🎯

Problem:

"How do you optimise temp table queries?"

Requirement:

→ Indexes! 🎯

Result: Temp tables/table variables appear in 65% of advanced SQL interviews!

SQL Mastery Journey

• Videos 1-3: Foundation (SELECT, JOIN, GROUP BY)

• Videos 4-5: Intermediate (Time, Subqueries)

• Video 6: CTEs (Readable SQL)

• Video 7: Window Functions (Rankings)

• Video 8: PIVOT (Matrix reports)

• Video 9: Stored Procedures (Automation)

• Video 10: Temp Tables & Variables ← COMPLEX WORKFLOWS MASTERS! 🎯

Next Video Preview

Video 11: Transactions & Error Handling

Commonwealth Bank - Data Integrity & ACID Principles

Duration: 18 minutes

Learn:

• BEGIN TRANSACTION, COMMIT, ROLLBACK

• ACID principles (Atomicity, Consistency, Isolation, Durability)

• Savepoints for partial rollback

• Transaction isolation levels

• Use temp tables for multi-step processing

• Return quarterly summary by account

• Deadlock handling

• Weekend handling

• Banking fund transfers (must be atomic!)

• Problem: Transfer \$1,000 from Account A to Account B

• Requirement: BOTH debit AND credit must succeed (or both fail)

• Solution: Transactions ensure data integrity

Result: Transactions & Error Handling

Commonwealth Bank - Data Integrity & ACID Principles

Duration: 18 minutes

Learn:

• BEGIN TRANSACTION, COMMIT, ROLLBACK

• ACID principles (Atomicity, Consistency, Isolation, Durability)

• Savepoints for partial rollback

• Transaction isolation levels

• Use temp tables for multi-step processing

• Return quarterly summary by account

• Deadlock handling

• Weekend handling

• Banking fund transfers (must be atomic!)

• Problem: Transfer \$1,000 from Account A to Account B

• Requirement: BOTH debit AND credit must succeed (or both fail)

• Solution: Transactions ensure data integrity

Result: Transactions & Error Handling

Commonwealth Bank - Data Integrity & ACID Principles

Duration: 18 minutes

PRACTICE EXERCISES & MASTERY SUMMARY

1. Basic Temp Table ★

- Create temp table #HighValueAccounts
- Insert accounts with opening_balance > \$50,000
- Query and display all columns
- Manually DROP the temp table

Hint: CREATE TABLE #Name (...); INSERT INTO #Name SELECT...

- ✓ Use temp tables descriptively (#Step1_Transactions)
- ✓ Document multi-step workflows with comments
- ✓ Wrap complex workflows in stored procedures
- ✓ Test with sample data before production
- ✓ Monitor tempdb usage in production