Session 9 – Stored Procedures: Concept & Syntax

[Begin Teleprompter-Style Script]

You're leading the backend for a financial product. Salary, bonuses, tax calculations — all happening in tight monthly windows. Do you want that logic duplicated across APIs, services, and frontend calls?

Of course not.

You want it centralized. Secure. Fast.

That's where ***Stored Procedures*** *come in.*

Think of them as the database equivalent of reusable JavaScript functions.

```
function calculateBonus(salary, rating) {
  if (rating === "A") return salary * 0.25;
  if (rating === "B") return salary * 0.15;
  return salary * 0.05;
}
```

Simple, right?

Now imagine this logic running directly inside the database — as a stored procedure.

So what ***is*** *a stored procedure?*

A stored procedure is a precompiled set of SQL statements — packaged together and stored inside the database.

You can invoke it with a single command, just like calling a function in JS.

Let's break down the benefits:

Modularity – You define it once and call it anywhere. **Reusability** – DRY principle, inside your DB layer. **Security** – You control access at the procedure level. No raw table access needed.

Now, here's a direct analogy for JS folks:

```
// In JS
function getUserProfile(id) {
  return db.query(`SELECT * FROM users WHERE user_id = ${id}`);
}
```

In SQL, this becomes:

```
CREATE PROCEDURE GetUserProfile(IN userId INT)
BEGIN
  SELECT * FROM users WHERE user_id = userId;
END;
```

Just call it like this:

```
CALL GetUserProfile(101);
```

One call. One clean result.

Now, let's make it real.

Say you want to calculate employee bonuses based on performance.

In JavaScript, you'd do something like:

```
function generateBonus(name, salary, rating) {
  const bonus = calculateBonus(salary, rating);
  console.log(`${name}'s bonus is ${bonus}`);
}
```

Let's imagine a stored procedure version of this.

```
CREATE PROCEDURE GenerateBonus(
  IN empName VARCHAR(50),
  IN empSalary DECIMAL(10,2),
  IN empRating VARCHAR(1),
  OUT empBonus DECIMAL(10,2)
)
BEGIN
  IF empRating = 'A' THEN
    SET empBonus = empSalary * 0.25;
  ELSEIF empRating = 'B' THEN
    SET empBonus = empSalary * 0.15;
  ELSE
    SET empBonus = empSalary * 0.05;
  END IF;
END;
```

We pass in the name, salary, and rating — and it calculates the bonus.

To call it:

```
CALL GenerateBonus('John', 80000, 'A', @bonus);
SELECT @bonus;
```

Boom — result in seconds, cleanly separated from app logic.

You might be asking — when do I really need this?

Here's my take after 15 years in the field:

- *When you need consistency across different services*
- *When you want to hide business logic from application developers*
- *When performance matters — stored procedures are precompiled*
- *And when you need tight access control*

Oh, and one last thing before we code it live…

*Stored procedures also support **IN**, **OUT**, and **INOUT** parameters — just like function arguments and return values in JS.*

Think of them as input args, return channels, or both.

Alright — next up, I'll walk you through building this live.

We'll modify the bonus logic interactively…

We'll tweak parameters…

And show you how small changes in logic don't need changes in the application — just the procedure.

Below are **working SQL examples** using your existing schema — `Customers`, `Products`, and `Orders` :

1. **User-Defined Function (UDF)**
2. **Stored Procedure**
3. **Trigger**

These examples are grounded in real-world enterprise use cases and are fully functional for your schema.

---

1. User-Defined Function (UDF)

Use Case: Calculate total amount of an order (price × quantity) **Used in**: Reporting, dashboards, financial calculations

```
-- Create UDF to calculate total price for an order
CREATE FUNCTION dbo.fn_CalculateOrderAmount
(
    @product_id INT,
    @quantity INT
)
```

```
RETURNS DECIMAL(10, 2)
AS
BEGIN
    DECLARE @price DECIMAL(10, 2)

    SELECT @price = price FROM Products WHERE product_id = @product_id

    RETURN @price * @quantity
END
GO

-- Example usage in a SELECT query
SELECT
    o.order_id,
    c.name AS customer_name,
    p.name AS product_name,
    o.quantity,
    dbo.fn_CalculateOrderAmount(o.product_id, o.quantity) AS total_amount
FROM Orders o
JOIN Customers c ON o.customer_id = c.customer_id
JOIN Products p ON o.product_id = p.product_id
```

---

2. Stored Procedure

Use Case: Generate customer invoice based on order ID **Goal**: Reusable, modular financial reporting

```
-- Create stored procedure to get order invoice
CREATE PROCEDURE dbo.sp_GetOrderInvoice
    @order_id INT
AS
BEGIN
    SELECT
        o.order_id,
        c.name AS customer_name,
        c.email,
        c.phone,
        p.name AS product_name,
        p.price,
        o.quantity,
        p.price * o.quantity AS total_amount,
        o.order_date
    FROM Orders o
    JOIN Customers c ON o.customer_id = c.customer_id
    JOIN Products p ON o.product_id = p.product_id
```

```
    WHERE o.order_id = @order_id
END
GO

-- Execute the stored procedure
EXEC dbo.sp_GetOrderInvoice @order_id = 1
```

---

3. Trigger

Use Case: Automatically log new order activity **Goal**: Auditing, automation

First, create an audit table:

```
-- Audit table to log order insertions
CREATE TABLE OrderAudit (
    audit_id INT IDENTITY(1,1) PRIMARY KEY,
    order_id INT,
    customer_id INT,
    action VARCHAR(50),
    audit_time DATETIME DEFAULT GETDATE()
)
```

Now create the trigger:

```
-- Trigger to log new orders into audit table
CREATE TRIGGER trg_AuditNewOrder
ON Orders
AFTER INSERT
AS
BEGIN
    INSERT INTO OrderAudit (order_id, customer_id, action)
    SELECT
        i.order_id,
        i.customer_id,
        'INSERT'
    FROM inserted i
END
GO

-- Test the trigger with an insert
INSERT INTO Orders (customer_id, product_id, quantity, order_date)
VALUES (1, 2, 3, GETDATE())

-- View audit
SELECT * FROM OrderAudit
```

Real-World Analogy

Think of a **trigger** like a fire alarm. You don't manually check for smoke — it just **responds instantly** when the condition is met.

Now imagine that for your database. • A customer deletes an order? Trigger can **archive it**. • A product's price changes? Trigger can **log it for audit**. • A new user registers? Trigger can **stamp the registration time**.

Automation — baked right into the data layer.

Use Case #1 – Archiving Deleted Orders

Let's say your boss asks: *"Hey, can we track which orders customers are deleting and when?"*

No problem. We'll build a trigger to do exactly that.

Step 1: Create the Archive Table

```
CREATE TABLE ArchivedOrders (
    order_id INT,
    customer_id INT,
    product_id INT,
    quantity INT,
    order_date DATE,
    deleted_at DATETIME DEFAULT GETDATE()
)
```

Simple mirror of the `Orders` table — with an extra `deleted_at` column for timestamp.

Step 2: Create the Trigger

```
CREATE TRIGGER trg_ArchiveDeletedOrders
ON Orders
AFTER DELETE
AS
BEGIN
    INSERT INTO ArchivedOrders (order_id, customer_id, product_id, quantity, order_date)
    SELECT
```

```
        order_id,
        customer_id,
        product_id,
        quantity,
        order_date
    FROM deleted
END
```

Here's what's happening:

- The keyword `AFTER DELETE` means: *"Run this AFTER a row is deleted from `Orders`."*
- The `deleted` table holds a snapshot of the deleted rows.
- We move them straight into `ArchivedOrders`.

Test It Out

```
-- Delete an order (simulate cancellation)
DELETE FROM Orders WHERE order_id = 1

-- Verify archived data
SELECT * FROM ArchivedOrders
```

Boom. You'll see the deleted order logged instantly.

Use Case #2 – Enforcing Business Rule (Min Order Quantity)

Now let's say you want to **prevent users from placing orders with quantity less than 1**. That's a business rule.

```
CREATE TRIGGER trg_PreventZeroQuantity
ON Orders
INSTEAD OF INSERT
AS
BEGIN
    IF EXISTS (
        SELECT * FROM inserted WHERE quantity <= 0
    )
    BEGIN
        RAISERROR('Order quantity must be at least 1', 16, 1)
        RETURN
    END

    -- Proceed with valid insert
    INSERT INTO Orders (customer_id, product_id, quantity, order_date)
```

```
    SELECT customer_id, product_id, quantity, order_date
    FROM inserted
END
```

Try inserting an order with `quantity = 0` — and the system will stop it, automatically.

Use Case #3 – Auto Timestamp Customer Update

Want to know *when* a customer updated their profile?

```
-- Add column to store update timestamp
ALTER TABLE Customers ADD last_updated DATETIME

-- Trigger to auto-update timestamp
CREATE TRIGGER trg_UpdateCustomerTimestamp
ON Customers
AFTER UPDATE
AS
BEGIN
    UPDATE Customers
    SET last_updated = GETDATE()
    FROM Customers c
    INNER JOIN inserted i ON c.customer_id = i.customer_id
END
```

Whenever someone changes their email or phone — `last_updated` updates silently. No dev intervention. No UI change needed.

Wrap-Up

So just to recap:

- **Triggers automate reactions** to INSERTs, UPDATEs, DELETEs.
- They're **event-driven**, not manually invoked.
- Perfect for **audit logging, business rules**, and **automated workflows**.

But here's the catch: **Use them sparingly and wisely**. Triggers are invisible to most devs and can cause confusion if abused. I've seen teams debug for hours — only to realize, *"Oh, there's a trigger firing silently…"*

Use triggers for what they're meant to do: **automate the things that should never be forgotten**.

Optimization #1: Indexes Are Your Best Friend

Imagine this — you're searching a phonebook. Do you flip every page, or go straight to the letter 'K' for "Kashyap"?

That's an index.

In SQL, without indexes, the engine scans every row — **called a full table scan**.

Let's start with a slow query.

Slow Query Example

```
SELECT *
FROM Customers
WHERE email = 'user@example.com'
```

This query will **scan the entire Customers table**, even if only one row matches.

Let's fix it.

Solution: Add an Index

```
CREATE INDEX idx_email ON Customers(email)
```

Now, the database can **jump straight to the row**, just like flipping to the right page in a book.

Re-run the same query — you'll see **instant improvement**.

> Pro Tip: Always index **high-cardinality columns** used in WHERE or JOINs.

Optimization #2: Use EXPLAIN to Understand the Engine

Don't guess. **Ask the engine** how it plans to execute your query.

```
EXPLAIN SELECT * FROM Customers WHERE email = 'user@example.com'
```

This returns a **query plan** — think of it as the engine's "GPS route."

If it says `type: ALL`, it's doing a full scan. You want to see things like `ref`, `range`, or `index`.

> Treat `EXPLAIN` like a debugger for performance.

Optimization #3: Avoid Unnecessary Subqueries

Now look at this:

```
SELECT name
FROM Customers
WHERE customer_id IN (
    SELECT customer_id FROM Orders WHERE quantity > 5
)
```

At a glance, this looks fine. But it's **inefficient** — especially with growing tables.

Rewrite with a JOIN

```
SELECT DISTINCT c.name
FROM Customers c
JOIN Orders o ON c.customer_id = o.customer_id
WHERE o.quantity > 5
```

Much cleaner. More efficient. The optimizer loves joins — not nested subqueries.

Optimization #4: Use Proper JOINs and Conditions

Let's take a use case where you want to pull product data for orders placed in the last 30 days.

Here's the right way:

```
SELECT o.order_id, c.name AS customer_name, p.name AS product_name, p.price
FROM Orders o
JOIN Customers c ON o.customer_id = c.customer_id
JOIN Products p ON o.product_id = p.product_id
WHERE o.order_date >= DATEADD(day, -30, GETDATE())
```

We're using:

- Inner joins
- Filter on indexed column (`order_date`)
- Clean SELECT fields (not `SELECT *`)

Optimization #5: Use LIMIT (or TOP) to Reduce Load

When you're building dashboards or paginated tables, **don't pull everything.** Just pull what you need.

```
SELECT TOP 10 * FROM Orders ORDER BY order_date DESC
```

Or with pagination:

```
SELECT * FROM Orders ORDER BY order_date DESC OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY
```

> Every millisecond saved here scales across your user base.

Demo: Optimize a Slow SELECT Query

Alright, let's say we want to get all customer names and products they've ordered in the last year.

Unoptimized Query

```
SELECT name
FROM Customers
WHERE customer_id IN (
    SELECT customer_id FROM Orders WHERE order_date >= '2024-01-01'
)
```

Looks fine, but subquery + no index = pain.

Optimized Version

1. Add an index:

```
CREATE INDEX idx_order_date ON Orders(order_date)
```

2. Rewrite with JOIN:

```
SELECT DISTINCT c.name
FROM Customers c
JOIN Orders o ON c.customer_id = o.customer_id
WHERE o.order_date >= '2024-01-01'
```

3. Use EXPLAIN:

```
EXPLAIN SELECT ...
```

You'll now see the engine using an index scan or range query — not a full scan.

Final Note: Balance Performance and Maintainability

As you optimize, keep two principles in mind:

1. **Know your data distribution.** Don't index everything — it costs write performance and storage.
2. **Use the right tool for the job.** Sometimes denormalization, materialized views, or even caching in code is better.

Wrap-Up

You just leveled up on five core query optimization practices:

- Use indexes on frequently filtered columns
- Leverage `EXPLAIN` to reveal performance problems
- Avoid subqueries when JOINs do better
- Keep your `SELECT` focused — not greedy
- Limit your data pulls and paginate when needed

In production, these aren't just tips — they're **survival strategies**. They keep your database lean, your APIs fast, and your users happy.

Simulating a Stored Procedure: Bonus Update (SQL Version)

Goal: Award a 10% bonus to customers who've ordered **more than 5 items** in total.

Assumptions:

- You have a column `bonus` in the `Customers` table (you'll need to add it if it doesn't exist).
- The bonus is applied only if the total quantity of orders by a customer exceeds 5.

Step 1: Add a `bonus` column to `Customers` (if not already present)

```
ALTER TABLE Customers
ADD bonus DECIMAL(5,2) DEFAULT 0;
```

Step 2: Create the Stored Procedure

```
CREATE PROCEDURE UpdateCustomerBonus
AS
```

```sql
BEGIN
    -- First, reset all bonuses
    UPDATE Customers
    SET bonus = 0;

    -- Then, award a 10% bonus to qualifying customers
    UPDATE Customers
    SET bonus = 10
    WHERE customer_id IN (
        SELECT customer_id
        FROM Orders
        GROUP BY customer_id
        HAVING SUM(quantity) > 5
    );
END;
```

Step 3: Execute the Stored Procedure

```sql
EXEC UpdateCustomerBonus;
```

Output Example

After running this, customers who've ordered more than 5 items in total will have a `bonus` value of `10`, and others will remain at `0`.

Insight:

This approach is optimal for batch updates. In a production system, you'd schedule this using a job (like SQL Server Agent, PostgreSQL cron, etc.) or trigger it after key events (e.g., new order).

Simulating a Trigger: Product Price Change Logging (SQL Version)

Step 1: Create the PriceChangeLog table

```sql
CREATE TABLE PriceChangeLog (
    log_id INT IDENTITY(1,1) PRIMARY KEY,
    product_id INT NOT NULL,
```

```
    old_price DECIMAL(10,2),
    new_price DECIMAL(10,2),
    changed_at DATETIME DEFAULT GETDATE()
);
```

Step 2: Create the Trigger on `Products` table

```
CREATE TRIGGER trg_ProductPriceChange
ON Products
AFTER UPDATE
AS
BEGIN
    -- Insert price changes into the log only when price actually changes
    INSERT INTO PriceChangeLog (product_id, old_price, new_price, changed_at)
    SELECT
        i.product_id,
        d.price AS old_price,
        i.price AS new_price,
        GETDATE()
    FROM inserted i
    INNER JOIN deleted d ON i.product_id = d.product_id
    WHERE i.price <> d.price;
END;
```

How It Works:

- This trigger runs **after every update** on the `Products` table.
- It compares the **old price** (from `deleted`) and **new price** (from `inserted`).
- If the price has changed, it logs the details to `PriceChangeLog`.

Example Usage:

```
-- Initial price is 1000
UPDATE Products
SET price = 1050
WHERE product_id = 1;

-- This change will be logged.

-- No price change
UPDATE Products
```

```
SET price = 50
WHERE product_id = 2;

-- This change will NOT be logged.
```

Real-World Takeaway:

In enterprise systems, this is critical for **auditing**, **compliance**, and **tracing**. Think of sectors like banking, healthcare, or e-commerce—triggers quietly enforce accountability behind the scenes.

Bonus Challenge (For Group Activity):

Want to log only if the price changes **by more than 20%**?

Update the `WHERE` clause like so:

```
WHERE ABS(i.price - d.price) / d.price > 0.20
```

That would log only *significant* price fluctuations.