Stored Procedures & Functions - Automation & Reusability

BHD Mining Operations Report Automation

Video 9 of 15 | Duration: 19 minutes | Level: Advanced

THE MONDAY MORNING FRUSTRATION

8:00 AM Monday - Operations Manager Walks In

> "I'm FRUSTRATED! Every single day I waste 2 HOURS running the same queries for our daily production report!"

The Daily Manual Nightmare

01

Step 1-3

Open Word document with 15+ saved queries → Copy first query → Paste into SSMS

02

Step 4-5

Manually change date: '2024-12-01' → '2024-12-02' → Run query

03

Step 6-7

Copy results to Excel → Format cells, add formulas

04

Step 8-10

Repeat steps 2-7 for NEXT query (14 more times!) → Compile all into one report → Email to stakeholders

2hrs	10hrs	520hrs	$80K
Time wasted	Weekly waste	Annual waste	Annual cost
EVERY DAY		= 13 WEEKS of work!	in analyst time

The Reports Needed Daily

1. Production Summary (ore tonnes by site and shift)
2. Equipment Downtime Analysis
3. Safety Incidents (by severity and status)
4. Efficiency Metrics (equipment utilisation %)
5. Site Comparisons (yesterday vs last week)
6. Shift Performance Rankings

... 10 more reports!

All require:

- Same data sources
- Same calculations
- Same formatting
- Just different **DATES!**

The "Copy-Paste-Change-Date" Hell

Problems:

- Manual date changes → Human errors
- Copy-paste mistakes → Wrong data in reports
- Inconsistent calculations → Different analysts calculate differently
- No version control → "Which query is the latest?"
- Time-consuming → 2 hours EVERY day

This is what Operations Manager deals with DAILY!

What If We Could...

- Press **ONE** button?
- Run **ALL** reports automatically?
- With **consistent** logic?
- In **30 seconds** instead of 2 hours?

That's what STORED PROCEDURES do!

The Solution: Automation with Stored Procedures

1

Instead of 15 separate queries in Word document:

Create **ONE** stored procedure

2

Instead of manual date changes:

Pass date as **PARAMETER**

3

Instead of 2 hours:

30 seconds execution time

4

Instead of $80K annual waste:

$75K+ annual SAVINGS

The Magic Command

Before (Manual):

Open Word → Copy Query 1 → Paste → Change date → Run → Export
Repeat 15 times → **2 hours**

After (Automated):

```
EXEC DailyOperationsReport @ReportDate = '2024-12-01'
```

30 seconds → Done! ✅

What You'll Master

Stored Procedures

- CREATE PROCEDURE with parameters
- Default parameters and NULL handling
- OUTPUT parameters (return multiple values)
- TRY...CATCH error handling
- Production-ready automation

Functions

- Scalar functions (return single value)
- Table-valued functions (return table)
- When to use functions vs procedures

Business Impact

- Automate repetitive workflows
- Build reusable code libraries
- 96% time savings (2 hours → 5 minutes)
- Error handling prevents failures

PLUS: 5 Practice Exercises with Solutions

The Monday Morning Victory

1 — 8:00 AM — "I need to automate this!"

2 — 10:00 AM — Stored procedure written

3 — 11:00 AM — Tested with last week's data

4 — 2:00 PM — Deployed to production

5 — Tuesday 8:00 AM — Operations Manager runs report in 30 seconds

6 — Tuesday 8:01 AM — "This is AMAZING! What took 2 hours now takes 30 seconds!"

Annual impact: 520 hours saved = You just gave them 13 weeks back!

STORED PROCEDURES - THE BASICS

Packaging SQL for Reuse

What Is a Stored Procedure?

Definition: Pre-compiled SQL code STORED in the database. Like creating a custom "program" that runs on demand. Accept inputs (parameters), perform logic, return results.

Think of it as:

Regular query = Recipe written on paper (write every time)

Stored procedure = Recipe saved in cookbook (reuse anytime)

Benefits:

- ✅ **Reusability** - Write once, use everywhere
- ✅ **Performance** - Pre-compiled, optimised execution
- ✅ **Security** - Grant EXECUTE permission, hide table access
- ✅ **Maintainability** - Change in one place, affects all uses
- ✅ **Consistency** - Same logic, same results every time

Basic Stored Procedure Syntax

```
CREATE PROCEDURE procedure_name
    @parameter1 datatype,
    @parameter2 datatype
AS
BEGIN
    -- Your SQL code here
    SELECT ...
    WHERE column = @parameter1
    ...
END
GO
```

Executing the Procedure

```
EXEC procedure_name @parameter1 = value1, @parameter2 = value2;
```

Or shorthand:

```
EXEC procedure_name value1, value2;
```

Real BHD Example - Get Site Production

```
CREATE PROCEDURE GetSiteProduction
    @SiteID INT
AS
BEGIN
    SELECT
        ms.site_name,
        dp.production_date,
        dp.shift,
        dp.ore_tonnes,
        dp.downtime_hours
    FROM daily_production dp
    JOIN mining_sites ms ON dp.site_id = ms.site_id
    WHERE dp.site_id = @SiteID
    ORDER BY dp.production_date DESC;
END
GO
```

Execute It

```
EXEC GetSiteProduction @SiteID = 1;
```

Result: All production data for Olympic Dam (Site 1)

```
EXEC GetSiteProduction @SiteID = 4;
```

Result: All production data for South Flank (Site 4)

Same procedure, different parameter → Different results!

Parameters – Making Procedures Flexible

@SiteID INT → INT = Integer (whole numbers)

Common data types:

- **INT** - Whole numbers (1, 2, 100)
- **DECIMAL(10,2)** - Decimals (123.45)
- **DATE** - Dates (2024-12-01)
- **NVARCHAR(100)** - Text up to 100 characters
- **BIT** - True/False (0 or 1)

Parameters are **INPUTS** to your procedure. Like function arguments in Python or Excel.

Multiple Parameters Example

```
CREATE PROCEDURE GetProductionByDateRange
    @StartDate DATE,
    @EndDate DATE,
    @SiteID INT
AS
BEGIN
    SELECT ...
    WHERE production_date BETWEEN @StartDate AND @EndDate
    AND site_id = @SiteID
    ...
END
GO
```

Call with 3 parameters:

```
EXEC GetProductionByDateRange
    @StartDate = '2024-12-01',
    @EndDate = '2024-12-07',
    @SiteID = 1;
```

Why Use Stored Procedures?

Scenario: Daily report query used by 5 analysts

Without Procedure:

- Each analyst has their own copy of the query
- Someone finds a bug → Must update 5 copies
- Calculation changes → Must update 5 copies
- Inconsistent results (different versions)

With Procedure:

- One stored procedure, everyone uses it
- Bug found → Fix once, affects everyone
- Calculation changes → Update once
- **Consistent results guaranteed!**

Procedures vs Regular Queries

Regular Query:

- ✓ Simple, one-time analysis
- ✓ Ad-hoc questions
- ✓ Quick exploration
- × Repetitive work
- × No reusability
- × No version control

Stored Procedure:

- ✓ Reusable automation
- ✓ Consistent business logic
- ✓ Parameter-driven flexibility
- ✓ Version controlled
- ✓ Production workflows
- × Overkill for one-time queries

When to Create a Procedure

Ask yourself:

- "Will I run this query again?" → **YES** → Procedure
- "Do others need this report?" → **YES** → Procedure
- "Does this query change slightly each time?" → **YES** → Parameterise it!
- "Is this mission-critical?" → **YES** → Procedure with error handling

Examples:

- Daily/weekly/monthly reports → **Procedure**
- One-time data exploration → Regular query
- Standard calculations (revenue, margins) → **Procedure**
- Ad-hoc stakeholder question → Regular query

The BHD Use Case

1 | 2

Before Procedure:

- 15 queries in Word document
- Manual execution daily
- 2 hours of work
- High error risk

After Procedure:

- 1 stored procedure: DailyOperationsReport
- EXEC with one date parameter
- 30 seconds execution
- Automated, error-free

Result: 96% time savings! 🎯

DEFAULT PARAMETERS & OUTPUT PARAMETERS

Advanced Procedure Features

Default Parameters - Making Parameters Optional

Problem: What if user doesn't provide a parameter? Force them to type every parameter every time? Annoying!

Solution: DEFAULT VALUES

```
CREATE PROCEDURE GetProductionByDateRange
    @StartDate DATE,
    @EndDate DATE = NULL,     ← Default to NULL
    @SiteID INT = NULL        ← Optional
AS
BEGIN
    -- If @EndDate not provided, default to today
    IF @EndDate IS NULL
        SET @EndDate = GETDATE();

    SELECT ...
    WHERE production_date BETWEEN @StartDate AND @EndDate
    AND (@SiteID IS NULL OR site_id = @SiteID)
    ...
END
GO
```

Now Can Call Multiple Ways

All parameters:

```
EXEC GetProductionByDateRange
    @StartDate = '2024-12-01',
    @EndDate = '2024-12-07',
    @SiteID = 1;
```

Without @EndDate (uses today):

```
EXEC GetProductionByDateRange
    @StartDate = '2024-12-01',
    @SiteID = 1;
```

Without @SiteID (all sites):

```
EXEC GetProductionByDateRange
    @StartDate = '2024-12-01';
```

Flexibility! User chooses what to filter!

The Optional Filter Pattern

```
WHERE (@SiteID IS NULL OR site_id = @SiteID)
```

- If @SiteID IS NULL → Condition TRUE for all rows (no filter)
- If @SiteID = 1 → Only rows where site_id = 1

This is a STANDARD pattern for optional filters in procedures!

Use it for:

- (@State IS NULL OR state = @State)
- (@Category IS NULL OR category = @Category)
- (@Status IS NULL OR status = @Status)

OUTPUT Parameters - Returning Calculated Values

Problem: Procedure runs calculations. How to GET those values back to the caller?

Solution: OUTPUT PARAMETERS

```
CREATE PROCEDURE GetSiteSummary
    @SiteID INT,
    @TotalOre DECIMAL(12,2) OUTPUT,       ← OUTPUT parameter
    @TotalIncidents INT OUTPUT,           ← OUTPUT parameter
    @DowntimePercent DECIMAL(5,2) OUTPUT   ← OUTPUT parameter
AS
BEGIN
    SELECT @TotalOre = SUM(ore_tonnes)
    FROM daily_production
    WHERE site_id = @SiteID;

    SELECT @TotalIncidents = COUNT(*)
    FROM safety_incidents
    WHERE site_id = @SiteID AND resolved = 0;

    SELECT @DowntimePercent =
        (SUM(downtime_hours) / SUM(equipment_hours)) * 100
    FROM daily_production
    WHERE site_id = @SiteID;
END
GO
```

Calling with OUTPUT Parameters

```
-- Step 1: Declare variables to receive values
DECLARE @Ore DECIMAL(12,2);
DECLARE @Incidents INT;
DECLARE @Downtime DECIMAL(5,2);

-- Step 2: Call procedure with OUTPUT keyword
EXEC GetSiteSummary
    @SiteID = 1,
    @TotalOre = @Ore OUTPUT,
    @TotalIncidents = @Incidents OUTPUT,
    @DowntimePercent = @Downtime OUTPUT;

-- Step 3: Use the returned values
PRINT 'Total Ore: ' + CAST(@Ore AS NVARCHAR(20));
PRINT 'Unresolved Incidents: ' + CAST(@Incidents AS NVARCHAR(10));
PRINT 'Downtime: ' + CAST(@Downtime AS NVARCHAR(10)) + '%';
```

Result:

```
Total Ore: 115240.50
Unresolved Incidents: 2
Downtime: 5.8%
```

When to Use OUTPUT Parameters

Use OUTPUT when:

- Need to return MULTIPLE calculated values
- Values are scalars (single numbers/strings)
- Calling from another procedure that needs the values
- Building multi-step workflows

Don't use OUTPUT when:

- Returning a result set (just SELECT)
- Only need one value (use RETURN instead)
- Values are tables (use table-valued function)

OUTPUT vs SELECT vs RETURN

Method 1: OUTPUT Parameters

Returns: Multiple scalar values

Use: Pass calculated values to calling code

Method 2: SELECT Statement

Returns: Result set (table of data)

Use: Display data to user, reports

Method 3: RETURN (status code)

Returns: Single integer

Use: Success/failure status (0 = success, 1+ = error)

Example combining all three:

```
CREATE PROCEDURE ProcessSiteData
    @SiteID INT,
    @ProcessedCount INT OUTPUT
AS
BEGIN
    -- Calculate (OUTPUT)
    SELECT @ProcessedCount = COUNT(*) FROM ...

    -- Display (SELECT)
    SELECT site_name, ore_tonnes FROM ...

    -- Status (RETURN)
    RETURN 0;  -- Success
END
GO
```

Real-World Use Case

Dashboard that needs:

- Total ore today
- Unresolved incidents count
- Average efficiency %

Without OUTPUT:

- Run 3 separate queries
- 3 round trips to database

With OUTPUT:

- One procedure call
- All 3 values returned
- One round trip
- **Faster!**

```
DECLARE @Ore DECIMAL(12,2), @Incidents INT, @Efficiency DECIMAL(5,2);

EXEC GetDashboardMetrics
 @SiteID = 1,
 @TotalOre = @Ore OUTPUT,
 @IncidentCount = @Incidents OUTPUT,
 @AvgEfficiency = @Efficiency OUTPUT;

-- Now use @Ore, @Incidents, @Efficiency in dashboard
```

ERROR HANDLING - PRODUCTION-READY CODE

TRY...CATCH for Robust Procedures

Why Error Handling Matters

Production scenario: User runs procedure with SiteID = 999 (doesn't exist)

- **Without error handling** → Query returns no rows, user confused
- **With error handling** → "Error: Site ID 999 not found" (clear!)

Consequences of no error handling:

- Silent failures (no data returned, user doesn't know why)
- Cryptic SQL error messages users don't understand
- Procedures crash mid-execution
- Data corruption (partial updates)

Error handling = Professional, production-ready code!

TRY...CATCH Syntax

```
BEGIN TRY
    -- Code that might fail
    SELECT ...
    UPDATE ...
    IF something_wrong
        THROW error
END TRY
BEGIN CATCH
    -- Handle the error
    PRINT error message
    Log to error table
    Re-throw or suppress
END CATCH
```

Real BHP Example - Safe Production Report

```
CREATE PROCEDURE GetSafeProductionReport
    @SiteID INT,
    @ReportDate DATE
AS
BEGIN
    BEGIN TRY
        -- Validation: Check inputs
        IF @SiteID IS NULL OR @SiteID <= 0
            THROW 50001, 'Invalid Site ID', 1;

        IF @ReportDate IS NULL
            THROW 50002, 'Report date required', 1;

        -- Validation: Check site exists
        IF NOT EXISTS (SELECT 1 FROM mining_sites WHERE site_id = @SiteID)
            THROW 50003, 'Site ID does not exist', 1;

        -- If validations pass, return data
        SELECT ...
        FROM daily_production
        WHERE site_id = @SiteID
        AND production_date = @ReportDate;

        PRINT 'Report generated successfully';

    END TRY
    BEGIN CATCH
        -- Capture error details
        DECLARE @ErrorMessage NVARCHAR(4000) = ERROR_MESSAGE();
        DECLARE @ErrorSeverity INT = ERROR_SEVERITY();

        -- Log error
        PRINT 'ERROR: ' + @ErrorMessage;

        -- Re-throw to caller
        THROW;
    END CATCH
END
GO
```

Testing Error Handling

Valid call (works fine):

```
EXEC GetSafeProductionReport
@SiteID = 1, @ReportDate =
'2024-12-01';
```

Result: Report data returned ✓

Invalid SiteID (triggers error):

```
EXEC GetSafeProductionReport
@SiteID = 999, @ReportDate =
'2024-12-01';
```

Result: "ERROR: Site ID does not exist" 🚫

NULL date (triggers error):

```
EXEC GetSafeProductionReport
@SiteID = 1, @ReportDate =
NULL;
```

Result: "ERROR: Report date required" 🚫

Users get CLEAR error messages!

THROW Statement - Raising Custom Errors

```
THROW error_number, 'error message', state;
```

- **error_number:** 50000+ (user-defined range)
- **error message:** Clear description for user
- **state:** Usually 1

Examples:

```
THROW 50001, 'Invalid Site ID', 1;
THROW 50010, 'Date cannot be in the future', 1;
THROW 50020, 'Site is not operational', 1;
```

You define the numbers, keep them consistent!

ERROR Functions - Capturing Error Details

Inside CATCH block, use:

- `ERROR_MESSAGE()` - Error description text
- `ERROR_SEVERITY()` - Error severity level (1-25)
- `ERROR_STATE()` - Error state (helps identify where it occurred)
- `ERROR_NUMBER()` - Error number
- `ERROR_LINE()` - Line number where error occurred
- `ERROR_PROCEDURE()` - Procedure name where error occurred

Example - Detailed Error Logging

```
BEGIN CATCH
    DECLARE @ErrorMsg NVARCHAR(4000) = ERROR_MESSAGE();
    DECLARE @ErrorSeverity INT = ERROR_SEVERITY();
    DECLARE @ErrorState INT = ERROR_STATE();
    DECLARE @ErrorLine INT = ERROR_LINE();

    -- Log to error table (in production)
    INSERT INTO error_log (error_message, severity, line_number, logged_date)
    VALUES (@ErrorMsg, @ErrorSeverity, @ErrorLine, GETDATE());

    -- Print for immediate feedback
    PRINT 'ERROR at line ' + CAST(@ErrorLine AS NVARCHAR(10));
    PRINT 'Message: ' + @ErrorMsg;

    -- Re-throw
    THROW;
END CATCH
```

Production Best Practices

1. **Validate ALL inputs**
   Check for NULL. Check for invalid ranges (negative numbers, future dates). Check for existence (does SiteID exist?)

2. **Use meaningful**