Datasets Queries Reference

Create reusable dataset queries.

Reference **Modules** **Datasets** **UI** DBs | Queries | Query Editor | Tables | Files | Monitor

---

Dataset Queries (Reference) enable SQL-based data operations including SELECT, INSERT, UPDATE, DELETE statements and stored procedure execution. Dataset Queries provide:

- SQL statement execution
- Real-time parameter embedding
- Tag-to-column mapping
- Stored procedure support
- Synchronous/asynchronous execution
- Result set visualization

Queries connect to configured DBs and can embed tag values dynamically in SQL statements.

Configuration Properties
Creating Queries
SQL Statement Configuration
Tag Mapping
Stored Procedures
Execution Methods
Diagnostic Properties
Query Editor
Visual Query Builder
Testing Queries
Performance Optimization
Best Practices Checklist
Troubleshooting
Related Documentation

Configuration Properties

Property	Description	Required
Name	Unique query identifier (no spaces)	Yes
DB	Target database connection	Yes
SqlStatement	SQL query or stored procedure	Yes
Mapping	Tag-to-column assignments	No
MappingDateTime	Time interpretation (UTC/Local)	No
Description	Documentation text	No

Creating Queries

1. Navigate to **Datasets Queries**
2. Click **Plus icon**
3. Configure:
    - **Name**: Unique identifier (use underscores)
    - **DB**: Select database
    - **Description**: Documentation
4. Click **OK**
5. Double-click **SqlStatement** to open editor

SQL Statement Configuration

Static Queries

sql

```sql
SELECT * FROM Production WHERE Status = 1
UPDATE Inventory SET Count = 100 WHERE ID = 5
DELETE FROM TempData WHERE Age > 30
```

Dynamic Parameters

Embed tag values using double curly braces:

sql

```sql
SELECT * FROM Production
WHERE BatchID = {{Tag.Production/Batch_ID}}
  AND Temperature > {{Tag.TankFarm/Tank1/Temp}}
  AND Status = {{Tag.Equipment/MotorRunning}}
```

DateTime Format

Default format: `yyyy-MM-dd HH:mm:ss` (ISO 8601)

sql

```sql
SELECT * FROM History
WHERE Timestamp > '{{Tag.StartDate}}'
```

Tag Mapping

Configure result set mapping to tags:

1. Click **Mapping** ellipsis (...)
2. Assign columns to tags:

Column	Tag	Direction
ProductID	Tag.Production_ID	Output
Count	Tag.Production_Count	Output
BatchID	Tag.Batch_ID	Input/Output

Mapping Options

- **Output**: Column value Tag
- **Input**: Tag Query parameter
- **Input/Output**: Bidirectional

Stored Procedures

Configuration

sql

```sql
-- Simple execution
EXEC GetProductionData @Date = '{{Tag.QueryDate}}'

-- With output parameters
EXEC TestOutput
  @return_Value = {{Tag.ReturnValue}} RETURN_VALUE,
  @vcrPrefix = @null@,
  @intNextNumber = {{Tag.NextNumber}} OUTPUT,
  @vcrFullLicense = {{Tag.NextLicense}} OUTPUT
```

Null Parameters

Use `@null@` syntax for null values:

sql

```sql
EXEC UpdateBatch @ID = {{Tag.BatchID}}, @Notes = @null@
```

Execution Methods

Asynchronous Execution (Default)

Trigger: Property changes (Select, Insert, Update, Delete)

Flow:

1. Property triggered (screen/script)

2. Request propagated to server
3. Dataset Module receives request
4. TServer executes database operation
5. Results returned to Dataset Module
6. Tags mapped and updated
7. Execution continues in parallel

Advantages:

- Non-blocking operation
- Better performance
- Prevents UI freezing
- Allows parallel operations

Use Cases:

- Display queries
- Background updates
- Report generation
- Real-time monitoring

Synchronous Execution

Trigger: Method calls (SelectCommand, ExecuteCommand)

Flow:

1. Method called
2. Execution PAUSES
3. Dataset Module calls TServer
4. Database operation completes
5. Results returned
6. Tags mapped
7. Execution RESUMES

Advantages:

- Guaranteed completion
- Sequential logic
- Immediate results
- Transaction support

Risks:

- Can freeze UI if called from screen
- Blocks thread execution
- Performance bottlenecks

Use Cases:

- Script tasks
- Sequential operations
- Transaction requirements
- Data validation

Code Examples

Asynchronous (Properties):

csharp

```
// Trigger query execution
@Dataset.Query.MyQuery.SqlStatement = "SELECT * FROM Data";
@Dataset.Query.MyQuery.Select = true;

// Check status
if (@Dataset.Query.MyQuery.LastStatus == 0)
{
    // Process results
}
```

Synchronous (Methods):

csharp

```
// Execute and wait for results
DataTable result = @Dataset.Query.MyQuery.SelectCommand();

// With status check
string status;
@Dataset.Query.MyQuery.ExecuteCommandWithStatus(out status);
```

Script Example

csharp

```
public void ExecuteQuery()
{
    // Build dynamic query
    string sql = string.Format(
        "SELECT * FROM Production WHERE Line = '{0}'",
        @Tag.ProductionLine
    );

    @Dataset.Query.ProductionQuery.SqlStatement = sql;

    // Execute with status
    string status;
    @Dataset.Query.ProductionQuery.ExecuteCommandWithStatus(out status);

    if (status == "OK")
    {
        @Tag.QueryResult = "Success";
    }
}
```

Diagnostic Properties

Monitor query execution with these properties:

```
@Dataset.Query.Query1.LastStatus        // Last status code (0 = success)
@Dataset.Query.Query1.LastStatusMessage // Text of the last status
@Dataset.Query.Query1.RowCount          // Result size
```

Query Editor

Built-in SQL editor features:

- Syntax highlighting
- Auto-completion
- Schema browsing
- Parameter validation

Access via:

- Double-click SqlStatement column
- Right-click Edit Query

Visual Query Builder

Graphical query construction:

1. Navigate to **Datasets DBs**
2. Click **Visual Query Builder**
3. Drag tables to canvas
4. Connect fields visually
5. Generate SQL automatically

Testing Queries

Data Explorer

1. Go to **Data Explorer SQL Queries**
2. Select query from list
3. Set parameter values

4. Click **Execute**
5. Review results in grid

Runtime Testing

csharp

```
// Test mode execution
@Dataset.Query.TestQuery.SqlStatement =
    "SELECT COUNT(*) FROM Production";
@Dataset.Query.TestQuery.Select = true;
```

Performance Optimization

Query Optimization

sql

```
-- Use parameters to prevent recompilation
SELECT * FROM Table WHERE ID = @id

-- Limit result sets
SELECT TOP 100 * FROM LargeTable

-- Use appropriate indexes
CREATE INDEX idx_timestamp ON Data(Timestamp)
```

Asynchronous Patterns

csharp

```
// Good: Asynchronous from screen
@Dataset.Table.MyTable.SelectCommand = "SELECT * FROM Data";

// Bad: Synchronous from screen (blocks UI)
@Dataset.Table.MyTable.SelectCommandWithStatus();
```

Batch Operations

- Group multiple operations
- Use transactions for consistency
- Minimize round trips

sql

```
-- Use transactions
BEGIN TRANSACTION
UPDATE Table1...
UPDATE Table2...
COMMIT
```

Best Practices Checklist

? **Use parameters** - Prevent SQL injection ? **Name meaningfully** - ProductionData not Query1 ? **Document queries** - Use description field ? **Test thoroughly** - Use Data Explorer ? **Handle errors** - Check execution status ? **Optimize performance** - Index, limit results ? **Use appropriate execution** - Async for UI ? **Prefer asynchronous** - Use properties over methods ? **Avoid UI blocking** - No synchronous calls from screens

---

Troubleshooting

Query not executing:

- Verify DB connection
- Check SQL syntax
- Confirm tag names
- Review permissions

Slow queries:

- Check execution plan
- Add appropriate indexes
- Reduce result set size

- Use asynchronous execution

Mapping issues:

- Verify column names
- Check tag types
- Review data types
- Confirm tag existence
- Verify domain selection
- Review mapping configuration
- Confirm execution context

Thread blocking:

- Avoid synchronous in UI
- Use Script Tasks
- Implement async patterns
- Monitor thread pool

Related Documentation

- Datasets DBs (Reference)
- Datasets Tables (Reference)
- Datasets Engine (Reference)

In this section...
