JDBC Adapters

# Understanding Transactions in webMethods

## No Transaction

**Scenario: Inserting Customer Data into a Database (Without Transaction)**

**Imagine you have an application that collects customer information and inserts it into a database. Each customer record is inserted separately, and failure of one record does not impact others.**

**How It Works?**

1. **Customer submits details**
   * **A new customer fills out a registration form with Name, Email, and Phone Number.**
2. **webMethods receives the data**
   * **webMethods takes the customer data and prepares an SQL INSERT statement.**
3. **Using No Transaction Mode**
   * **webMethods executes the JDBC INSERT operation using No Transaction mode.**
   * **Each record is inserted immediately without waiting for other records.**
4. **Insertion completes immediately**
   * **If one record fails due to an issue (e.g., duplicate email), other records are not affected and still get inserted.**

**Example SQL Statements for Inserting Customers**

**For each customer, webMethods sends separate INSERT requests like this:**

**INSERT INTO Customers (CustomerID, Name, Email, Phone) VALUES (1, 'John Doe', 'john@example.com', '9876543210');**

**INSERT INTO Customers (CustomerID, Name, Email, Phone) VALUES (2, 'Alice Smith', 'alice@example.com', '9123456789');**

**INSERT INTO Customers (CustomerID, Name, Email, Phone) VALUES (3, 'Bob Johnson', 'bob@example.com', '9988776655');**

**Each statement is executed individually and committed immediately.**

**Key Points About No Transaction Mode**

**✔ Each INSERT is a separate operation.  
✔ If one record fails, others continue inserting.  
✔ Faster execution since there is no rollback handling.**

## Local Transaction

**In webMethods, transaction management is primarily handled by the database, while webMethods acts as a bridge to initiate and control transactions using JDBC adapter services. The database ensures that changes are either fully committed or completely rolled back in case of failure.**

**How Transactions Work?**

1. **Start a transaction → The process begins when webMethods initiates a transaction.**
2. **Execute multiple database operations → Multiple insert, update, or delete operations are performed within the same transaction.**
3. **Success or Failure:** 
   * **If all operations succeed, the commit command is executed, making the changes permanent.**
   * **If any operation fails, the rollback command is triggered, undoing all changes made in that transaction.**

**Real-Time Example: Employee Registration System**

**Imagine an HR system where a company registers new employees. The process involves inserting data into two related tables:**

1. **Employee Table → Stores general employee details.**
2. **Employee\_Salary Table → Stores salary details for each employee.**

**Since both tables should be updated together, a Local Transaction ensures consistency.**

**Successful Case (Commit Scenario)**

**Steps in the Process:**

1. **Insert into Employee Table → The basic details of the employee are inserted successfully.**
2. **Insert into Employee\_Salary Table → The salary details for the employee are also inserted successfully.**
3. **Commit Triggered → Since both operations are successful, changes are permanently saved in the database.**

**Example SQL Statements in Local Transaction:**

**BEGIN TRANSACTION; -- Transaction starts**

**INSERT INTO Employee (EmpID, Name, Age, Department) VALUES (101, 'John Doe', 30, 'IT');**

**INSERT INTO Employee\_Salary (EmpID, Salary, Bonus) VALUES (101, 60000, 5000);**

**COMMIT; -- Commit transaction since all operations succeeded**

**✅ Result: The employee and salary records are successfully inserted into the database.**

**Failure Case (Rollback Scenario)**

**Steps in the Process:**

1. **Insert into Employee Table → The basic details of the employee are inserted successfully.**
2. **Insert into Employee\_Salary Table → A delay occurs (e.g., due to a database constraint or network issue), and the insertion fails.**
3. **Rollback Triggered → The previously inserted employee record is also removed to maintain consistency.**

**Example SQL Statements in Local Transaction:**

**BEGIN TRANSACTION; -- Transaction starts**

**INSERT INTO Employee (EmpID, Name, Age, Department) VALUES (102, 'Alice', 28, 'HR');**

**INSERT INTO Employee\_Salary (EmpID, Salary, Bonus) VALUES (102, 70000, NULL); -- This fails due to NULL constraint**

**ROLLBACK; -- Rollback transaction since one operation failed**

**❌ Result: The employee record is not saved because the transaction failed, ensuring data consistency.**

**Where is Data Stored Before Commit?**

**Before committing a transaction, the data is stored temporarily in the database’s transaction log or buffer cache:**

* **Undo Logs / Rollback Segments → Helps roll back the changes if needed.**
* **Memory (Buffer Cache) → Holds uncommitted data temporarily.**
* **Not visible to other transactions/users → Until the transaction is committed.**

**Can a DB Admin See This Temporary Data?**

**No regular user can see this uncommitted data, but DB administrators might track it through internal logs and monitoring tools.**

XA TransactionFlipkart Order Processing

**Scenario:**  
Imagine you are placing an order on **Flipkart** that includes multiple payment and inventory operations:

1. **Payment Processing** (stored in the PaymentsDB).
2. **Order Entry** (stored in the OrdersDB).
3. **Stock Deduction** (stored in the InventoryDB).

If any one of these fails, the entire order should be **rolled back** to maintain consistency.

**How XA Transaction Works in webMethods?**

**Step 1: Start the XA Transaction**

* webMethods starts an **XA Transaction** across multiple databases.

**Step 2: Execute the Operations**

1. **Insert Order Details**
   * Insert the order information into the **OrdersDB**.
2. **Process Payment**
   * Deduct the amount from the customer’s account in **PaymentsDB**.
3. **Update Inventory**
   * Reduce the stock count in **InventoryDB**.

**Step 3: Two-Phase Commit Process**

1. **Prepare Phase:**
   * Each database confirms it is ready to commit the changes.
2. **Commit Phase:**
   * If all databases respond positively, webMethods **commits** the changes.
   * If any one operation fails (e.g., payment gateway issue), **rollback** is triggered for all.

**How Rollback Happens?**

* If payment **fails**, the order is **removed**, and stock remains unchanged.
* If inventory update **fails**, the payment is refunded, and the order is deleted.
* Until commit, changes are **only in temporary storage (transaction log)**, ensuring that no incomplete data is saved permanently.

**Key Benefits of XA Transaction in Flipkart**

✔ Ensures **data consistency** across multiple systems.  
✔ Prevents cases like **money deducted but order not placed**.  
✔ Supports **multiple database rollbacks automatically**.

Let me know if you need more details!

**webMethods’ Role in Transactions**

* **Defines transaction types (Local, XA, or No Transaction).**
* **Controls transaction boundaries using startTransaction, commitTransaction, and rollbackTransaction.**
* **Uses JDBC Adapter Services to interact with the database.**

**Key Takeaways**

1. **webMethods doesn’t handle transactions internally – it only manages them at a high level while the database performs actual commit/rollback.**
2. **Until a transaction is committed, data is stored in the database’s temporary memory and is not visible to others.**
3. **Rollback ensures data integrity by removing any partial updates if an error occurs.**

**This approach guarantees data consistency and prevents data corruption in case of failures.**

# WebMethods JDBC Adapters - Connection Pooling

### 1. What is Connection Pooling?

Connection pooling is a mechanism that **reuses database connections** instead of creating a new connection for every request. This improves performance and reduces the overhead of frequently opening and closing connections.

### 2. How Connection Pooling Works in WebMethods?

1. When an **adapter service** executes, it requests a connection from the **connection pool**.
2. If a connection is **available**, it is assigned to the request.
3. If no connection is available:
   * If the **maximum pool size** is not reached, a new connection is created.
   * If the **maximum pool size** is reached, the request waits in a queue until a connection is released.
4. After execution, the connection is **returned to the pool** for reuse.

### 3. Connection Pool Settings in WebMethods

| **Property** | **Description** |
| --- | --- |
| **Minimum Pool Size** | The number of connections **always kept open**, even if unused. |
| **Maximum Pool Size** | The **maximum number of connections** that can be created. |
| **Pool Increment Size** | If no free connections are available, **this many new connections** are created. |
| **Block Timeout (ms)** | The time a request waits for a connection if none are available. |
| **Expire Timeout (ms)** | The time an **idle connection** stays in the pool before being closed. |

### 4. Example: Handling Multiple Requests

**Scenario**

* JDBC Adapter Connection:
  + **Min Pool Size = 1**
  + **Max Pool Size = 3**
* Four users (A, B, C, D) call the **same adapter service** at different times.

**Execution Timeline**

|  |  |  |  |
| --- | --- | --- | --- |
| **Time** | **User** | **Connection Used** | **Status** |
| 10:00:00 AM | User A | Conn-1 | **Running** |
| 10:00:01 AM | User B | Conn-2 | **Running** |
| 10:00:02 AM | User C | Conn-3 | **Running** |
| 10:00:03 AM | User D | **Waiting** | **Queued** |
| 10:00:04 AM | User A finishes | Conn-1 released |  |
| 10:00:04 AM | User D gets Conn-1 | **Running** |  |

**Key Takeaways**

* **Users A, B, and C get connections immediately**.
* **User D waits** because all connections are in use.
* **Once a request completes, its connection is returned to the pool**.

### 5. Why Do We Need Multiple Connections?

* If only **one connection** is available, every request must **wait** for the previous one to finish.
* Multiple connections allow **parallel execution**, avoiding delays.
* Helps **efficiently handle multiple user requests** without blocking.

### 6. What Happens If the Max Pool is Reached?

* If all connections are **in use**, new requests **must wait**.
* If a request waits longer than the **Block Timeout**, it **fails** with an error.
* Once a connection is **released**, waiting requests **get assigned a free connection**.

**Conclusion**

* Connection pooling **improves performance** by reusing database connections.
* Proper **tuning of Min/Max Pool Size** ensures **efficient DB utilization**.
* If too **few connections**, requests **queue up and cause delays**.
* If too **many connections**, it may **overload the database**.

This document summarizes **how webMethods JDBC Adapter Connection Pooling works** and how to **optimize it for better performance**.

# SQL Injection in webMethods JDBC Adapter Services

### 1. What is SQL Injection?

SQL Injection is a security vulnerability that allows attackers to manipulate SQL queries by injecting malicious input. This can lead to unauthorized data access, modification, or even deletion of database records.

### 2. How Does SQL Injection Happen in webMethods?

When dynamic SQL queries are constructed improperly in webMethods, an attacker can modify query logic by inserting malicious input. This typically happens in **Dynamic SQL Template** adapter services when user inputs are directly concatenated into the SQL statement.

**Example of Vulnerable Query:**

SELECT \* FROM Employee WHERE EmpID = " + param1

If param1 = 101 OR 1=1, the final query becomes:

SELECT \* FROM Employee WHERE EmpID = 101 OR 1=1

Since 1=1 is always **true**, all employee records will be retrieved, leading to data exposure.

### 3. Preventing SQL Injection in webMethods

To prevent SQL injection, follow these best practices:

**A. Use Proper Adapter Services**

* **Use Select, Insert, Update, Delete, StoredProcedure Templates** instead of DynamicSQL Template.
* These templates use placeholders (?) that safely bind variables, preventing SQL injection.

**B. Use Input Parameter Binding**

Example of a **safe query** using SelectSQL template:

SELECT \* FROM Employee WHERE EmpID = ?

Here, ? is replaced securely with the parameter value, preventing injection attacks.

**C. Validate User Input**

* Ensure input values are properly validated before passing them to queries.
* Reject inputs containing SQL keywords (e.g., DROP, DELETE, OR 1=1).

**D. Use Stored Procedures**

* Stored procedures **encapsulate SQL logic** within the database, making it harder to exploit injection vulnerabilities.
* Example:

CALL GetEmployeeDetails(?)

### 4. Conclusion

SQL Injection is a serious security risk in webMethods if adapter services are not designed properly. Using proper SQL templates, parameter binding, input validation, and stored procedures ensures secure database interactions.

By following these best practices, webMethods developers can prevent SQL injection attacks and enhance application security.

# JDBC Adapter Templates – Definitions & Real-World Examples

## 1. SelectSQL Template

**Definition**

Retrieves specific data from one or more tables in a database based on SQL SELECT queries.

**Real-World Example**

A telecom company wants to retrieve customer details where the customer’s status is “Active.” You use **SelectSQL** to query the Customers table for active users.

## 2. InsertSQL Template

**Definition**

Inserts a new row into a database table using a simple or parameterized INSERT statement.

**Real-World Example**

When a customer orders a new SIM card, their order details need to be inserted into the Orders table. **InsertSQL** can handle this automatically.

## 3. UpdateSQL Template

**Definition**

Updates existing rows in a database table based on specific criteria.

**Real-World Example**

When a customer changes their address, you use **UpdateSQL** to update the Address field in the Customers table where CustomerID = ?.

## 4. DeleteSQL Template

**Definition**

Deletes rows from a database table based on specific filter conditions.

**Real-World Example**

In a telecom system, when a prepaid customer account is closed, **DeleteSQL** removes their data from the PrepaidAccounts table.

## 5. BatchInsertSQL Template

**Definition**

Inserts multiple rows efficiently in a batch operation.

**Real-World Example**

A marketing team uploads a CSV of new promotional offers. **BatchInsertSQL** inserts all offers into the Promotions table in a single batch.

## 6. BatchUpdateSQL Template

**Definition**

Updates multiple rows in a batch using a document list.

**Real-World Example**

During tariff updates, the system needs to update rates for hundreds of plans. **BatchUpdateSQL** updates all plans in a single batch.

## 7. CustomSQL Template

**Definition**

Executes any SQL statement (including complex joins or custom queries) that doesn’t fit standard templates.

**Real-World Example**

Generating a monthly report that joins Orders, Customers, and Payments with conditions on multiple fields. **CustomSQL** executes this complex query.

## 8. DynamicSQL Template

**Definition**

Executes SQL with dynamically built parts (like WHERE clauses) at runtime.

**Real-World Example**

A customer search screen where filters like name, location, and status are optional. **DynamicSQL** dynamically builds the SQL query based on input filters.

## 9. StoredProcedure Template

**Definition**

Calls a stored procedure in the database, with manually defined input/output parameters.

**Real-World Example**

Calling an ApplyDiscount stored procedure that calculates a discount based on customer loyalty. **StoredProcedure** triggers this procedure.

## 10. StoredProcedureWithSignature Template

**Definition**

Similar to StoredProcedure, but retrieves the procedure’s signature (input/output parameters) automatically.

**Real-World Example**

Calling an UpdateInventory procedure with multiple input/output parameters without manually defining them. **StoredProcedureWithSignature** introspects the procedure signature.

## 11. ExecuteService Template

**Definition**

Executes a custom Java or Flow service while using a JDBC connection from the adapter’s pool.

**Real-World Example**

You need to run a custom Java service that performs a complex transaction involving multiple SQL statements. **ExecuteService** provides the JDBC connection to the Java service.

## Summary Table

|  |  |  |
| --- | --- | --- |
| **Template** | **Definition** | **Real-World Example** |
| SelectSQL | Retrieve data using SELECT. | Fetch customer details. |
| InsertSQL | Insert a new row. | Insert a new order. |
| UpdateSQL | Update existing rows. | Update customer address. |
| DeleteSQL | Delete rows. | Remove closed accounts. |
| BatchInsertSQL | Insert multiple rows in batch. | Import promotional offers. |
| BatchUpdateSQL | Update multiple rows in batch. | Update tariff plans. |
| CustomSQL | Execute complex/custom SQL. | Monthly reports with joins. |
| DynamicSQL | SQL with dynamic query parts. | Flexible customer search. |
| StoredProcedure | Call a stored procedure. | Apply discounts. |
| StoredProcedureWithSignature | Call stored procedure (auto-fetch signature). | Update inventory. |
| ExecuteService | Run a Java/Flow service with JDBC connection. | Complex custom transactions. |

## Difference Between CustomSQL and DynamicSQL in webMethods

**1. CustomSQL**

**What is it?**

* You define the **entire SQL statement** at design time (when you create the adapter service).
* You can use ? placeholders for input parameters, but the structure of the query (like table name, columns, WHERE clause) is fixed.

**Real-World Example:**

Suppose you have a table CustomerDetails where you want to select records based on a **customer ID**.

* Query:

SELECT \* FROM CustomerDetails WHERE CustomerID = ?

You create a **CustomSQL adapter service** with this SQL, and you specify an input field for CustomerID.

**When to Use?**

* Use when the SQL structure is known and fixed.
* Example: Fetch records, update specific rows, or run standard queries where only values change (like CustomerID = ?).

**2. DynamicSQL**

**What is it?**

* Part (or all) of the SQL statement is **set dynamically at runtime** based on input parameters.
* This means you can build queries with variable **WHERE clauses**, **column names**, or even entire SQL statements.

**Real-World Example:**

Suppose you have a **search screen** where a user can filter by multiple optional fields: name, location, and status.

* Query at design time:

SELECT \* FROM CustomerDetails ${whereClause}

At runtime, whereClause could be:

* WHERE name='John'
* or WHERE status='Active' AND location='India'
* or empty (no filter).

This **DynamicSQL** allows flexibility in building queries based on user input.

**When to Use?**

* Use when the query changes based on runtime data.
* Example: Flexible search functionality where filters are optional or vary.

**Simple Comparison Table**

| **Feature** | **CustomSQL** | **DynamicSQL** |
| --- | --- | --- |
| SQL Structure | Fixed at design time | Partially or fully dynamic at runtime |
| Use of Parameters | Uses ? placeholders | Uses ${fieldName} for dynamic injection |
| Flexibility | Low | High |
| Real Example | SELECT \* FROM Table WHERE ID = ? | SELECT \* FROM Table ${whereClause} |
| When to Use | Fixed queries | Flexible/optional queries |

**Practical Decision Example**

Let’s say you’re working for a **telecom system**:

* For **retrieving customer bills for a specific customer ID**, use **CustomSQL** because the query is fixed.
* For **building a dynamic report** where the filters depend on the user (e.g., location, date range), use **DynamicSQL**.

# JDBC Adapter Performance Optimization – Real-Time Example

**Example Scenario:**

You are working for a **telecom company**. The system receives **daily updates of customer billing records** from external systems (e.g., prepaid recharge data, postpaid bill payments). These records need to be **inserted into the Billing database**.

Initially, you implemented a **simple InsertSQL** operation for each record. However, under high load, the system started to slow down, and database connections were exhausted.

Let’s optimize this setup step-by-step.

## 1. Use Connection Pooling

* Instead of opening and closing a connection for each record, enable **connection pooling**.
* **Configuration**:
  + Minimum Pool Size = 5
  + Maximum Pool Size = 50
  + Pool Increment Size = 5
  + Block Timeout = 5000 ms (wait 5 seconds if no connection is available)
* **Result**: The system reuses existing connections, reducing overhead and connection errors.

## 2. Switch from InsertSQL to BatchInsertSQL

* Rather than executing an InsertSQL for each record, switch to **BatchInsertSQL**, which can insert multiple records in one call.
* Example: Suppose you receive 1000 billing records daily.
* **BatchInsertSQL** can insert all 1000 records in a single batch.
* **Result**: Reduces network round trips and speeds up processing significantly.

## 3. Set Query Timeout

* Sometimes, the database might slow down or lock. Set a **Query Timeout** so the operation won’t hang indefinitely.
* **Configuration**:
  + Query Timeout = 30 seconds.
* **Result**: If a database issue occurs, the process will fail gracefully and allow retries or alerts.

## 4. Choose the Right Transaction Type

* For batch operations like this, use **LOCAL\_TRANSACTION** to ensure all records in the batch are committed together.
* **Result**: Ensures consistency (all-or-nothing) for the entire batch of records.

## 5. Optimize the SQL Query

* Avoid unnecessary columns in the Insert or Select queries.
* Use bind variables (e.g., ? placeholders) to avoid parsing overhead and SQL injection.
* **Result**: Cleaner and safer queries.

## 6. Monitor and Tune

* Regularly check the **connection pool stats** (number of active, idle, and failed connections).
* Tune pool sizes based on observed load.
* Monitor database performance (e.g., long-running queries, locks).

## Summary of Optimization

* Step 1: Enable and tune connection pooling.
* Step 2: Replace single InsertSQL with BatchInsertSQL.
* Step 3: Configure QueryTimeout for reliability.
* Step 4: Use LOCAL\_TRANSACTION for consistency.
* Step 5: Write efficient SQL.
* Step 6: Monitor system health and tune configurations.

# ✅ Types of Transaction in JDBC Adapter

There are **three transaction types** available in JDBC Adapter:

## 1. NO\_TRANSACTION

* **What it is**: No explicit transaction boundaries.
* **How it works**:
  + Each adapter service runs as a standalone unit.
  + Once executed, changes are immediately committed to the database.
* **Use case**:
  + When you're only reading data (e.g., SELECT queries).
  + Small, isolated updates or inserts where rollback isn’t required.
* **Real-time example**:
  + Fetching customer details to display in UI.

## 2. LOCAL\_TRANSACTION

* **What it is**: A transaction within a single database connection.
* **How it works**:
  + Multiple adapter services within the same flow can participate in one transaction.
  + commit and rollback must be handled explicitly (usually in the success or failure sequence).
* **Use case**:
  + When you're performing multiple INSERT/UPDATE/DELETE operations on the same DB and want atomicity.
* **Real-time example**:
  + Inserting order header and line items into the same database.

## 3. XA\_TRANSACTION (Global or Distributed Transaction)

* **What it is**: A transaction that spans **multiple resources** (e.g., multiple DBs, JMS, etc.).
* **How it works**:
  + Uses the **Two-Phase Commit (2PC)** protocol to ensure all resources either commit or roll back together.
* **Use case**:
  + Required when one transaction involves more than one system (e.g., DB + JMS).
  + Ensures consistency across systems.
* **Real-time example**:
  + Updating an order in Oracle DB and sending a confirmation message via JMS (UM) in one transaction.

**💡 When to Use What?**

| **Transaction Type** | **Use Case** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| NO\_TRANSACTION | Read-only or isolated ops | Simple, fast | No rollback |
| LOCAL\_TRANSACTION | Single DB, atomic operations | Supports rollback | Can’t span across systems |
| XA\_TRANSACTION | Multi-system transaction (DB + JMS) | Full rollback support | Heavy and complex |

Absolutely Sankar! Here's your **complete and well-structured notes on JDBC Notifications in webMethods**, perfect for interviews and clear understanding.

# ✅ JDBC Notifications in webMethods

## 🔷 What are JDBC Notifications?

**JDBC Notifications** are used in webMethods to **monitor changes in a database** (like insert, update, or delete) and **trigger services automatically** in Integration Server when such events occur.

This is achieved using the **webMethods JDBC Adapter**, which acts as a bridge between Integration Server and the database.

## 🔷 Why JDBC Notifications?

* To **automate integration** when data changes in DB
* To avoid **manual polling** of the DB
* To build **event-driven architecture**

## 🔷 Types of JDBC Notifications

webMethods provides the following types of notifications:

| **Notification Type** | **Description** |
| --- | --- |
| **Insert Notification** | Triggered when a row is inserted into a table. |
| **Update Notification** | Triggered when a row is updated. |
| **Delete Notification** | Triggered when a row is deleted. |
| **Insert/Update/Delete Notification** | Monitors multiple actions (combined) on a table. |
| **Basic Notification** | Uses custom SQL to poll and check for a condition. |
| **Stored Procedure Notification** | Uses stored procedures to notify Integration Server. |

## 🔷 How It Works (for Insert/Update/Delete):

1. webMethods creates:
   * A **trigger** in the DB table.
   * A **buffer table** to temporarily store changes.
2. When a DB change occurs (e.g., row inserted), the DB trigger **copies that row into the buffer table**.
3. webMethods **polls the buffer table** and **invokes a flow service** when a new record is found.

## 🔷 Explanation of Each Notification Type:

### 🔸 1. Insert Notification

* Triggered when a row is **inserted**.
* Creates a **trigger** and **buffer table**.
* Flow service is triggered with the new row data.

📌 Use case: When a new order is placed, trigger order confirmation service.

### 🔸 2. Update Notification

* Triggered when an **existing row is updated**.
* Monitors the update operation via DB trigger.

📌 Use case: Trigger a flow when status='Shipped' is updated in an orders table.

### 🔸 3. Delete Notification

* Triggered when a **row is deleted**.
* Helps track deletions and take appropriate action.

📌 Use case: Audit deletions from a sensitive table like user\_roles.

### 🔸 4. Insert/Update/Delete (IUD) Notification

* Supports **multiple events** on the same table.
* Sends event type (Insert/Update/Delete) along with data.

📌 Use case: Monitor complete lifecycle of a transaction table.

### 🔸 5. Basic Notification

* **No trigger or buffer table**.
* You provide a **custom SQL SELECT query**.
* webMethods **polls** the DB using this query.

📌 Use case: Check for failed payments or retry count logic.

### 🔸 6. Stored Procedure Notification

* Relies on a stored procedure.
* DB must **call the procedure manually** when needed.
* Suitable when logic is encapsulated inside the database.

📌 Use case: DB team triggers notifications after complex batch processing.

## 🔷 Important Concepts

| **Term** | **Description** |
| --- | --- |
| **Buffer Table** | Temporary table where DB trigger inserts data. |
| **Polling Interval** | Time period between two DB checks by Integration Server. |
| **Trigger** | DB object created by webMethods to track changes. |
| **Flow Service** | Service invoked when notification is fired. |

## 🔷 Advantages of JDBC Notifications

✅ Real-time or near real-time integration  
✅ No need for manual DB checks  
✅ Reduced complexity in polling logic  
✅ Event-driven design  
✅ Decouples systems

## 🔷 Limitations

❌ DB structure modification (triggers/buffer tables) may be restricted by DB admins  
❌ High frequency polling may load the DB  
❌ Works only with RDBMS supported by JDBC Adapter (e.g., Oracle, SQL Server, DB2, etc.)

## 🔷 Best Practices

* Use **Basic Notification** for flexible, read-only checks.
* Use **Insert/Update/Delete Notification** for reliable and real-time triggers.
* **Avoid complex joins** in Basic Notification queries.
* Set **reasonable polling intervals** to reduce DB load.
* Log errors in the target flow service for debugging.