DB-Independent Dynamic CSV Ingestion Framework

Audience: Backend / Platform Engineers
Goal: Build a production-grade, DB-agnostic ingestion framework that can handle **unknown CSV schemas**, **schema drift**, **multiple RDBMS**, and **future Delta Lake writes** using **Spring Batch**.

---

1. Problem Statement

Business uploads CSV files with **unknown and changing schemas**: - Today: `master_data.csv` with 10 columns - Tomorrow: same file with 3 new columns - Later: data type change (STRING → INT)

Target systems: - MySQL - PostgreSQL - Oracle - SQL Server - (Future) Delta Lake

Key constraints: - Restartable ingestion - Safe schema evolution - No vendor lock-in - Production observability

---

2. High-Level Solution

We use **Spring Batch as an orchestrator**, not as a DB abstraction.

Core ideas: - Canonical schema (DB-neutral) - Dialect-based DDL/DML generation - Pluggable writers (RDBMS / Delta) - Step-wise execution (DDL before DML)

---

3. High-Level Architecture

```
                ┌──────────────────────────┐
                │     Spring Batch Job     │
                └──────────────────────────┘
                            │
        ┌───────────────────┼────────────────────┐
        │                   │                    │
        ▼                   ▼                    ▼
┌──────────────┐    ┌────────────────┐    ┌────────────────┐
│ Schema       │    │ Schema           │    │ Data Load        │
│ Discovery    │──▶ │ Management       │──▶ │ (Insert/Update)  │
│ (CSV)        │    │ (DDL)            │    │ (DML)            │
└──────────────┘    └────────────────┘    └────────────────┘
```

```
          │                    │                    │
          ▼                    ▼                    ▼
   Canonical Schema    Dialect-Based SQL     Writer Abstraction
                                                     │
                       ┌─────────────────────────────┼──────────────────────────┐
                       ▼                             ▼                          ▼
                     MySQL                       PostgreSQL                   Delta
Lake
```

4. Canonical Schema (Foundation)

All schemas are converted into a **canonical representation**.

4.1 Canonical Types

```
STRING | INTEGER | LONG | DECIMAL | BOOLEAN | DATE | TIMESTAMP
```

4.2 Canonical Column Model

```
class CanonicalColumn {
    String name;
    CanonicalType type;
    Integer length;
    Integer precision;
    Integer scale;
    boolean nullable;
}
```

Why this matters: - Shields business logic from DB specifics - Makes Delta Lake mapping trivial - Enables safe schema comparison

5. Step 1 – Schema Discovery (CSV)

Purpose

- Discover column names
- Infer data types
- Detect schema drift

How it works

1. Read CSV header
2. Sample first N rows
3. Infer canonical types
4. Store schema in `ExecutionContext`

Flow

```
CSV File
   │
   ▼
Read Header ──▶ Infer Types ──▶ Canonical Schema
                                  │
                                  ▼
                          ExecutionContext
```

Notes

- This step is DB-independent
- No table access here
- Safe to re-run

---

6. Step 2 – Schema Management (DDL)

Purpose

- Create table if missing
- Add new columns
- Modify existing columns (controlled)

Dialect Pattern

```
interface DatabaseDialect {
    String createTableSql(String table, List<CanonicalColumn> cols);
    List<String> alterTableSql(
        String table,
        List<CanonicalColumn> csvSchema,
        List<CanonicalColumn> dbSchema
    );
    String upsertSql(String table, Set<String> columns);
}
```

Each DB has its own implementation: - MySQLDialect - PostgresDialect - OracleDialect - SqlServerDialect

Flow

```
Canonical Schema
      │
      ▼
Load DB Metadata
      │
      ▼
Compare Schemas
      │
      ▼
Generate DDL (Dialect)
      │
      ▼
Execute DDL (Transactional)
```

Safety Rules (Recommended)

- ❌ Never auto-drop columns
- ❌ No narrowing conversions (STRING → INT)
- ✅ Allow widening (INT → STRING)
- ✅ Feature-flag ALTER operations

---

7. Step 3 – Data Load (Insert / Update)

Challenges

- Unknown columns
- DB-specific UPSERT syntax
- Batch performance

Solution

- Read rows as `Map<String, Object>`
- Generate SQL dynamically using dialect
- Batch using JDBC or Spark (Delta)

Flow

```
CSV Rows
  │
  ▼
Map<String,Object>
  │
  ▼
Dialect-Based UPSERT SQL
  │
  ▼
Batch Write
```

8. Writer Abstraction

Unified Contract

```
interface DataWriter<T> {
    void write(List<T> items);
}
```

RDBMS Writer

- Uses JDBC batch
- Uses dialect UPSERT SQL
- Fully transactional

Delta Lake Writer

- Uses Spark
- `mergeSchema = true`
- Append / merge supported

This allows **writer swap without job changes**.

9. Job Configuration Overview

```
Job: dynamicCsvIngestionJob

Step 1: schemaDiscoveryStep
Step 2: schemaSyncStep
Step 3: dataLoadStep
```

Each step: - Commits independently - Restartable - Auditable

10. Restartability & Failure Handling

Spring Batch automatically stores: - Last processed row - Step execution status - Failure reason

Restart Scenario

```
Failure at row 1,200,000
          |
          ▼
Restart Job
          |
          ▼
Resumes from last committed chunk
```

No manual tracking required.

---

11. Observability & Auditing

Recommended tables: - `batch_job_execution` - `batch_step_execution` - `ingestion_file_audit`

Track: - File name - Schema hash - Row count - Start / end time - Target system

---

12. Why Spring Batch (Final Justification)

Capability	JDBC Batch	Spring Batch
Dynamic schema	⚠️ Manual	✅ Structured
Restartability	❌	✅
Multi-DB	❌	✅
DDL orchestration	❌	✅
Delta Lake ready	❌	✅
Observability	❌	✅

---

13. Recommended Next Enhancements

- Schema versioning table

- Staging tables + merge
- File partitioning (parallelism)
- Prometheus metrics
- Iceberg / Hudi writers

---

14. Summary

This framework: - Handles schema drift safely - Is DB-agnostic by design - Scales from RDBMS to Lakehouse - Is production-ready and extensible

Spring Batch = Orchestrator
Dialect = Portability
Canonical Schema = Stability

End of document