Scala & Spark ETL — Complete Study Material (Beginner → Advanced)

Goal: Teach you how to design, implement, test, and operate robust ETL pipelines using Scala + Apache Spark. Includes beginner-friendly explanations, code snippets, illustrations, and recommended GitHub repositories to clone and study.

---

1. Table of contents

---

2. Overview & Learning Path

- **Phase 1 (Basics):** Scala syntax, sbt, small Spark jobs, read/write CSV/Parquet.
- **Phase 2 (Intermediate):** Transformations, joins, window functions, CTE-like logic using DataFrame APIs, UDFs/UDAFs.
- **Phase 3 (Advanced):** Optimize jobs (partitioning, broadcast joins), streaming (Structured Streaming), orchestration (Airflow), production practices (logging, metrics, retries).

Estimated time: 4–8 weeks of focused learning (self-paced).

---

3. Prerequisites & Tooling

Languages & frameworks: Scala (2.12.x or 2.13.x), Apache Spark 3.x

Tools:

- sbt (build tool)
- Git
- PostgreSQL (or any RDBMS) for sinks
- Docker (optional, for uniform environment)
- IDE: IntelliJ IDEA (Community) + Scala plugin
- Local Spark: start Spark with `spark-submit` or use `sbt run` for small apps

Recommended IDE settings:

- Enable Scala plugin, set SDK to Java 11 or 17 depending on Spark build.

---

4. Getting Started — Hands-on Setup

Install tools (brief)

- Install **Java 11** or 17.
- Install **sbt**: follow sbt docs or package manager.
- Install **PostgreSQL** (or use SQLite for simple tests).
- Install **Git** and clone repositories.

Sample project scaffold (sbt)

`build.sbt` minimal:

```
name := "simple-etl"

version := "0.1.0"

scalaVersion := "2.12.17"

libraryDependencies ++= Seq(
"org.apache.spark" %% "spark-core" % "3.4.1",
"org.apache.spark" %% "spark-sql" % "3.4.1",
"org.postgresql" % "postgresql" % "42.6.0"
)

assembly / mainClass := Some("com.example.etl.Main")
```

Adjust Spark and Scala versions if you use a corporate cluster. Keep Spark 3.x for modern compatibility.

Project layout

```
simple-etl/
├── src/main/scala/com/example/etl/
│   ├── Main.scala
│   ├── config/ConfigReader.scala
│   ├── jobs/CsvToParquetJob.scala
│   └── utils/SparkSessionBuilder.scala
├── src/test/scala/
├── build.sbt
└── conf/application.conf
```

---

5. Minimal ETL Example: CSV → Parquet → PostgreSQL

This example shows a readable, runnable template you can clone and expand.

SparkSession builder (utils/SparkSessionBuilder.scala)

```
package com.example.etl.utils

import org.apache.spark.sql.SparkSession

object SparkSessionBuilder {
def build(appName: String = "simple-etl") = {
SparkSession.builder()
.appName(appName)
.master("local[*]")
.config("spark.sql.shuffle.partitions", "4")
.getOrCreate()
}
}
```

Main entry (Main.scala)

```
package com.example.etl

import com.example.etl.jobs.CsvToParquetJob
import com.example.etl.utils.SparkSessionBuilder

object Main {
def main(args: Array[String]): Unit = {
```

```
val spark = SparkSessionBuilder.build("csv-to-parquet")
CsvToParquetJob.run(spark, args)
spark.stop()
}
}
```

Job: read CSV, transform, write Parquet and Postgres (jobs/CsvToParquetJob.scala)

```
package com.example.etl.jobs

import org.apache.spark.sql.{DataFrame, SparkSession}
import org.apache.spark.sql.functions._

object CsvToParquetJob {

def run(spark: SparkSession, args: Array[String]): Unit = {
val input = if (args.nonEmpty) args(0) else "data/input/invoices.csv"
val parquetOut = if (args.length > 1) args(1) else "data/output/invoices.parquet"

val df = readCsv(spark, input)
.withColumn("invoice_amount", col("quantity") * col("unit_price"))
.withColumn("invoice_date", to_date(col("invoice_date"), "yyyy-MM-dd"))

df.write.mode("overwrite").parquet(parquetOut)

writeToPostgres(df, "jdbc:postgresql://localhost:5432/accounting", "invoices")
}

def readCsv(spark: SparkSession, path: String): DataFrame = {
spark.read
.option("header", "true")
.option("inferSchema", "true")
.csv(path)
}

def writeToPostgres(df: DataFrame, jdbcUrl: String, table: String): Unit = {
val props = new java.util.Properties()
props.setProperty("user", "postgres")
props.setProperty("password", "postgres")

df.write
.mode("append")
.jdbc(jdbcUrl, table, props)
}
```

}

How to run locally

1. Put a sample CSV at `data/input/invoices.csv` (small sample included below).
2. `sbt run` or `sbt "run data/input/invoices.csv data/output/invoices.parquet"`

Sample CSV (invoices.csv)

invoice_id,customer_id,invoice_date,quantity,unit_price
1,100,2024-07-01,2,150.00
2,101,2024-07-03,1,200.00
3,100,2024-07-05,3,50.00

---

6. Project Structure & Best Practices

- **Separation of concerns:** Keep `config`, `io`, `transform`, `jobs` directories.
- **Idempotency:** Jobs should be re-runnable (use *write modes*, staging areas, or check markers).
- **Config-driven:** Use `application.conf` or environment variables for DB creds and paths.
- **Logging:** Use `log4j` or `slf4j`. Avoid println for production.
- **Small Tasks:** Design small, testable transformations.
- **Use DataFrame/Dataset API** instead of RDDs for declarative optimizations.

Example config (application.conf)

app {
env = "local"
input.path = "data/input"
output.path = "data/output"
jdbc.url = "jdbc:postgresql://localhost:5432/accounting"
}

---

7. Testing & CI

- Unit tests using `scalatest` or `munit`.
- Use `spark-testing-base` or create `SparkSession` with local master for tests.

Sample test

```
import org.scalatest.funsuite.AnyFunSuite
import com.example.etl.utils.SparkSessionBuilder

class CsvToParquetJobTest extends AnyFunSuite {
val spark = SparkSessionBuilder.build("test")

test("invoice_amount is calculated") {
import spark.implicits._
val df = Seq((1,2,100.0)).toDF("invoice_id","quantity","unit_price")
val withAmount = df.withColumn("invoice_amount", col("quantity") * col("unit_price"))
assert(withAmount.collect().head.getAs[Double]("invoice_amount") == 200.0)
}
}
```

CI tips:

- Run `sbt test` in CI (GitHub Actions). Use matrix for Java/Scala versions.
- Optionally build Docker image with `sbt assembly` and `spark-submit` for integration tests.

---

8. Advanced Topics & Performance

Partitioning & Bucketing

- Partition by `date` (e.g., `year=`, `month=`) for large fact tables.
- Use bucketing for join-heavy datasets with fixed key cardinality.

Write partitioned parquet

```
df.write.partitionBy("year", "month").parquet("/data/facts/invoices")
```

Broadcast joins

Use `broadcast(df)` for small dimension tables.

```
import org.apache.spark.sql.functions.broadcast
val joined = fact.join(broadcast(dim), Seq("id"))
```

Caching & Persisting

Cache intermediate DataFrames when reused multiple times:

```
val cached = df.filter(...).cache()
cached.count() // materialize
```

File formats

- Use Parquet for columnar efficiency.
- Consider **Delta Lake** or **Apache Hudi** for ACID + schema evolution.

Monitoring & Metrics

- Push job metrics to Prometheus (via Spark metrics or JVM exporter).
- Emit custom counters using `org.apache.spark.SparkContext.longAccumulator`.

---

9. Production Concerns: Orchestration, Observability, Security

Orchestration

- Use **Apache Airflow** (Python) to schedule and orchestrate Spark jobs. Provide a sample DAG to call `spark-submit` or submit jobs to a cluster.

Sample Airflow DAG (concept)

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime

with DAG('etl_pipeline', start_date=datetime(2025,1,1), schedule_interval='@daily') as dag:
run_etl = BashOperator(
task_id='run_spark_job',
bash_command='spark-submit --class com.example.etl.Main /opt/etl/simple-etl.jar'
)
```

Observability

- Centralized logs (ELK/EFK).
- Track job duration, input row counts, error counts.
- Alert when job fails or input counts deviate from baseline.

Security

- Use secrets manager for DB credentials (Vault, AWS Secrets Manager).
- Limit JDBC user privileges.
- TLS between services where possible.

---

10. Recommended GitHub Repositories (Beginner → Advanced)

Clone these to study patterns and sample pipelines.

Beginner / Learning

- `SETL (Scala ETL framework)` — github.com/SETL-Framework/setl
  - Why: Clean, modular; good to learn structure.
- `etl-spark` — github.com/alexland/etl-spark
  - Why: Minimalistic, good for a first real pipeline.
- `MyDataFramework` — github.com/vbounyasit/MyDataFramework
  - Why: Framework-oriented reusable pipelines.

Intermediate

- `spark-etl-framework` — github.com/qwshen/spark-etl-framework
  - Why: Focused on end-to-end ingestion → transformation.
- `spark-etl` — github.com/aphp/spark-etl
  - Why: Integrates Spark with Postgres/DB sinks.

Advanced / Production

- `Teams-League-Airflow-Spark-Scala-ETL` — github.com/tosun-si/teams-league-airflow-spark-scala-etl
  - Why: Shows orchestration + cloud storage + BigQuery patterns.
- `Scala-and-Spark-in-Practice` — github.com/ruslanmv/Scala-and-Spark-in-Practice-
  - Why: Exercises and performance patterns.

---

11. Mini Projects & Exercises

1. **Invoice ETL** — CSV invoices → clean → Parquet partitioned by year/month → load to Postgres. Add tests.

2. **Revenue Dashboard Feed** — Aggregate monthly revenue and write a materialized table.
3. **Duplicate Detector** — Find and report duplicate invoices (same customer, amount, date).
4. **Streaming variant** — Use Structured Streaming to ingest new transactions and update a running ledger.
5. **Data Quality Framework** — Create checks (row counts, null ratio, unique key checks) and store results.

---

12. Appendix: Useful Commands & Snippets

SBT assembly

sbt clean assembly

Build fat JAR and run with spark-submit

spark-submit --class com.example.etl.Main --master local[*] target/scala-2.12/simple-etl-assembly-0.1.0.jar data/input/invoices.csv data/output/invoices.parquet

**Export to CSV from Postgre