1) SQL — from basics to advanced

Save and run these on any PostgreSQL / MySQL / other SQL engine (small syntax differences noted).

a) Basic SELECT, WHERE, ORDER BY, LIMIT

```
-- Select specific columns from employees
SELECT id, name, salary                    -- choose id, name, salary
columns
FROM employees                             -- table employees
WHERE salary > 50000                       -- filter rows where salary
> 50000
ORDER BY salary DESC                       -- order descending by
salary
LIMIT 10;                                  -- return only top 10 rows
```

b) Joins (INNER, LEFT)

```
-- Inner join employees with departments
SELECT e.id, e.name, d.name AS dept_name   -- pick columns and alias
dept name
FROM employees e                           -- employees alias e
INNER JOIN departments d                   -- join departments alias d
  ON e.dept_id = d.id;                     -- join condition on
department id

-- Left join example (keep all employees even if no department)
SELECT e.id, e.name, d.name AS dept_name
FROM employees e
LEFT JOIN departments d
  ON e.dept_id = d.id;
```

c) Aggregation + GROUP BY + HAVING

```
-- Get average salary by department with filter on avg salary
```

```
SELECT d.name AS dept_name,                -- department name
       COUNT(*) AS emp_count,              -- number of employees
       AVG(e.salary)::numeric(10,2) AS avg_salary -- average salary
FROM employees e
JOIN departments d ON e.dept_id = d.id
GROUP BY d.name                             -- group rows per
department
HAVING AVG(e.salary) > 60000;               -- only groups where avg >
60k
```

d) Window functions (ROW_NUMBER, PARTITION BY)

```
-- Rank employees by salary within each department
SELECT id, name, dept_id, salary,
       ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary DESC)
AS rn
FROM employees
-- later you can filter for top-per-department using rn = 1
;
```

e) CTEs and subqueries

```
-- Use a CTE to compute high earners then filter
WITH high_earners AS (                      -- common table
expression
  SELECT id, name, dept_id, salary
  FROM employees
  WHERE salary > 80000
)
SELECT he.id, he.name, d.name AS dept_name
FROM high_earners he
JOIN departments d ON he.dept_id = d.id;
```

---

2) Scala — essentials (small runnable examples)

These are plain Scala snippets. You can paste into IntelliJ as small `object` files or run in REPL.

a) Variables, functions, basic types

```
object ScalaBasics {
  def main(args: Array[String]): Unit = {
    // immutable value
    val name: String = "Dani"            // declare a String constant
    // mutable variable
    var count: Int = 10                  // declare an Int variable
    count = count + 1                    // modify variable
    // simple function
    def add(a: Int, b: Int): Int = a + b // function returning a+b
    val sum = add(count, 5)              // call function
    println(s"Name: $name, Sum: $sum")   // print results
  }
}
```

b) Collections and functional ops (map, filter, flatMap)

```
object ScalaCollections {
  def main(args: Array[String]): Unit = {
    val nums = List(1,2,3,4,5)                      // List of Ints
    val evens = nums.filter(_ % 2 == 0)             // keep even
numbers
    val squares = nums.map(x => x*x)                // square each
number
    val pairs = nums.flatMap(n => List(n, n*10))    // expand each
into two values
    println(evens)                                  // print evens
    println(squares)
    println(pairs)
  }
}
```

c) Case classes, pattern matching, Option

```
case class Person(id: Int, name: String, age: Int) // immutable data
holder

object ScalaPattern {
  def safeGetName(optPerson: Option[Person]): String = optPerson match
{
    case Some(Person(_, name, _)) => name            // if present
return name
    case None => "unknown"                            // else default
  }

  def main(args: Array[String]): Unit = {
    val p = Person(1, "Arun", 28)
    println(safeGetName(Some(p)))                    // prints Arun
    println(safeGetName(None))                       // prints unknown
  }
}
```

---

3) Spark (Scala) — DataFrame examples with line-by-line comments

Below are Spark apps. Put them in an SBT project with Spark dependencies and run with Java 11 (as discussed earlier).

a) Simple Spark app: read CSV, transform, write Parquet

```
import org.apache.spark.sql.SparkSession                          //
import SparkSession

object SimpleSparkApp {
  def main(args: Array[String]): Unit = {
    // create SparkSession running locally using all cores
```

```
    val spark = SparkSession.builder()
      .appName("SimpleSparkApp")                                    // set
app name
      .master("local[*]")                                           // run
locally
      .getOrCreate()                                                //
instantiate session

    import spark.implicits._                                        //
bring implicits for toDF/as

    // create small DataFrame from sequence of tuples
    val df =
Seq((1,"Dani",30),(2,"Arun",28),(3,"Kumar",35)).toDF("id","name","age"
)
    df.show()                                                      // show
data to console

    // filter rows where age > 30
    val filtered = df.filter($"age" > 30)
    filtered.show()                                                //
display filtered results

    // write to parquet in local folder "output"

filtered.write.mode("overwrite").parquet("output/filtered.parquet")

    spark.stop()                                                   // stop
Spark session
  }
}
```

b) Spark SQL + temp view + UDF

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.udf

object SparkSqlUdf {
```

```
  def main(args: Array[String]): Unit = {
    val spark =
SparkSession.builder().appName("SparkSqlUdf").master("local[*]").getOr
Create()
    import spark.implicits._

    // sample DF
    val df =
Seq((1,"Dani",1000.0),(2,"Dev",2000.5)).toDF("id","name","salary")
    df.createOrReplaceTempView("emps")                          //
create SQL temporary view

    // define UDF to convert salary to string with currency
    val currencyUdf = udf((s: Double) => f"₹$s%.2f")          //
define udf to format salary
    spark.udf.register("currency", currencyUdf)               //
register UDF for SQL usage

    // use SQL with UDF
    val res = spark.sql("SELECT id, name, currency(salary) AS
salary_str FROM emps")
    res.show()                                                // show
results
    spark.stop()
  }
}
```

c) Spark Window functions example

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._

object SparkWindowExample {
  def main(args: Array[String]): Unit = {
    val spark =
SparkSession.builder().appName("SparkWindowExample").master("local[*]"
).getOrCreate()
```

```
    import spark.implicits._

    // sample sales: (dept, emp, sales)
    val sales = Seq(
      ("A","emp1",1000),
      ("A","emp2",1500),
      ("B","emp3",2000),
      ("A","emp4",1200),
      ("B","emp5",1800)
    ).toDF("dept","emp","sales")

    // define partition by dept and order by sales desc
    val w = Window.partitionBy($"dept").orderBy($"sales".desc)

    // compute row_number and rank within partition
    val ranked = sales.withColumn("rn", row_number().over(w))
                      .withColumn("rank", rank().over(w))
    ranked.show()                                                  // print
ranking results

    spark.stop()
  }
}
```

d) Broadcast join & optimization hint

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.broadcast

object BroadcastJoinExample {
  def main(args: Array[String]): Unit = {
    val spark =
SparkSession.builder().appName("BroadcastJoinExample").master("local[*
]").getOrCreate()
    import spark.implicits._

    // small lookup table - safe to broadcast
```

```
    val small =
Seq((1,"HR"),(2,"Engineering")).toDF("dept_id","dept_name")
    val large = (1 to 1000).map(i => (i, if(i%2==0) 1 else 2,
s"name_$i")).toDF("id","dept_id","name")

    // broadcast small to avoid shuffle join
    val joined = large.join(broadcast(small), "dept_id")
    joined.show(5)                                          // show
first 5 rows

    spark.stop()
  }
}
```

e) Example ETL pipeline (read JSON from S3-local, transform, write Parquet)

> Replace `s3a://...` with S3 path and configure Hadoop AWS credentials in real runs.

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

object EtlPipeline {
  def main(args: Array[String]): Unit = {
    val spark =
SparkSession.builder().appName("EtlPipeline").master("local[*]").getOr
Create()
    import spark.implicits._

    // read JSON file (local or s3 path)
    val raw = spark.read.option("multiLine",
true).json("data/input.json")

    // Basic cleaning: drop null id, trim strings
    val cleaned = raw.filter($"id".isNotNull)
                     .withColumn("name", trim($"name"))
                     .withColumn("ingest_ts", current_timestamp())
```

```
    // business logic: compute new column
    val enriched = cleaned.withColumn("salary_in_k", $"salary" / 1000)

    // write partitioned parquet

enriched.write.mode("overwrite").partitionBy("country").parquet("outpu
t/parquet")

    spark.stop()
  }
}
```

---

4) Spark Structured Streaming (simple source → sink)

This example reads JSON files appearing in a folder (file source) and writes aggregated counts to console.

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

object StreamingExample {
  def main(args: Array[String]): Unit = {
    val spark =
SparkSession.builder().appName("StreamingExample").master("local[*]").
getOrCreate()
    import spark.implicits._

    // read streaming data from JSON files landing in folder
"stream_in"
    val streamDF = spark.readStream
      .schema(Encoders.product[Event].schema)  // optional: provide
schema to avoid inference overhead
```

```
      .json("stream_in")                        // folder to watch for
new files

    // simple aggregation: count events by type in a sliding window
    val agg = streamDF.groupBy($"eventType", window($"ts", "1
minute"))
                      .count()

    // write stream to console in update mode
    val query = agg.writeStream
      .outputMode("update")
      .format("console")
      .option("truncate","false")
      .start()

    query.awaitTermination()
  }

  // case class for schema (example)
  case class Event(id: String, eventType: String, ts:
java.sql.Timestamp)
}
```

---

5) Data Quality checks (Spark snippets)

Simple DQ checks as Spark jobs — null checks, uniqueness, referential check.

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

object DataQualityChecks {
  def main(args: Array[String]): Unit = {
    val spark =
SparkSession.builder().appName("DataQualityChecks").master("local[*]")
.getOrCreate()
    import spark.implicits._
```

```
    val df =
spark.read.option("header","true").csv("data/customers.csv")
    // null check on id
    val nullCount = df.filter($"id".isNull).count()         // count
rows missing id
    println(s"Null id count: $nullCount")

    // uniqueness check on id
    val total = df.count()
    val distinctIds = df.select($"id").distinct().count()
    println(s"Total: $total, Distinct IDs: $distinctIds")

    // referential integrity: check if orders have customer ids
present in customers
    val orders =
spark.read.option("header","true").csv("data/orders.csv")
    val missingCustomers = orders.join(df, orders("cust_id") ===
df("id"), "left_anti").count()
    println(s"Orders with missing customers: $missingCustomers")

    spark.stop()
  }
}
```

---

6) Scala + Spark Integration: using Datasets & case classes

```
import org.apache.spark.sql.SparkSession

case class Person(id: Int, name: String, age: Int)   // case class
schema

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

```
    val spark =
SparkSession.builder().appName("DatasetExample").master("local[*]").ge
tOrCreate()
    import spark.implicits._

    val ds = Seq(Person(1,"A",30), Person(2,"B",25)).toDS() // create
Dataset[Person]
    ds.filter(_.age > 26).show()                            // use
typed lambda to filter and show
    spark.stop()
  }
}
```

---

7) Useful shell / sbt notes (to run in IntelliJ)

- Ensure `build.sbt` includes:

```
ThisBuild / scalaVersion := "2.12.18"
libraryDependencies += "org.apache.spark" %% "spark-core" % "3.5.1"
libraryDependencies += "org.apache.spark" %% "spark-sql"  % "3.5.1"
```

- Use **Java 11** for running Spark 3.x.
- Right-click `object` with `main` and Run in IntelliJ, or use `sbt run`.