Lab 1.3.2 — Distributed Batch Processing and Partitioning

Session 1.3 — Distributed Systems Fundamentals
Hands-on lab
Tools: Scala 3 (`scala-cli`), `batchlib.scala` (provided)

---

1. Objective

You will use a toy MapReduce framework to run distributed batch computations on a simulated cluster. The framework provides **workers** (actors that hold data and execute map/reduce functions) and a **cluster** (that manages workers and simulates network latency). You drive the pipeline — load, map, shuffle, reduce — from client code, choosing your own partitioning strategies and observing their effects on performance.

By the end you will understand why the **shuffle** dominates distributed batch runtime, why **data skew** kills parallelism, and how partitioning strategy shapes everything.

2. The Framework (`batchlib.scala`)

2.1. Protocol

Workers accept four commands:

- `LoadData(records, replyTo)` — store a chunk of records locally.
- `RunMap(mapFn, replyTo)` — apply `mapFn` to every local record, return `(key, value)` pairs.
- `ReceiveShuffle(data, replyTo)` — receive grouped `(key → values)` from the shuffle phase.
- `RunReduce(reduceFn, replyTo)` — apply `reduceFn` to each key's values, return final results.

2.2. Cluster

```
val cluster = Cluster("my-cluster", numWorkers = 4)
```

The cluster spawns `numWorkers` workers, each reachable by id (`w0`, `w1`, …). All communication goes through the cluster, which adds random 50–500ms network delay.

- `cluster.ask[Res](workerId, ref => command)` — send a command to one worker, get a `Future[Res]`.
- `cluster.askAll[Res](ref => command)` — broadcast the same command to all workers.
- `cluster.workerIds` — the list of worker ids.

2.3. Driving the pipeline

You orchestrate the four phases in `lab.scala`:

1. **Load** — partition your dataset, send each chunk to a worker via `LoadData`.

2. **Map** — send `RunMap(mapFn, ...)` to all workers. Each applies `mapFn` to its local data.
3. **Shuffle** — collect all map outputs, group by `hash(key) % numWorkers`, send to the right worker via `ReceiveShuffle`.
4. **Reduce** — send `RunReduce(reduceFn, ...)` to all workers. Merge the partial results.

3. Part 1 — Run the pipeline

3.1. 1.1 — Read and run `lab.scala`

The provided `lab.scala` generates 10,000 sales records and runs a full map-reduce pipeline that sums sales amounts by region. Run it:

```
scala-cli run lab.scala
```

Read the output. For each phase, note the wall-clock time and per-worker distribution.

3.2. 1.2 — Change the map/reduce functions

Modify the pipeline to compute a **count of sales per product** instead of sum by region. You only need to change `mapFn` and `reduceFn` — the framework stays the same.

Hint: your `mapFn` should emit `(product, "1")` for each record. Your `reduceFn` should count (or sum) the values.

3.3. 1.3 — Single-node baseline

Run the same computation on a single worker (`numWorkers = 1`). Compare the total time. Is 4 workers ~4× faster? Why not?

Discussion: the network delay dominates. With simulated 50–500ms per message, the actual computation (parsing CSV, summing ints) is negligible. In real systems the bottleneck shifts to disk I/O and network bandwidth, but the principle is the same: framework overhead vs. useful work.

4. Part 2 — Partitioning strategies

4.1. 2.1 — Round-robin loading (baseline)

The provided code loads data round-robin: chunk the dataset into equal parts, send one chunk per worker. Every worker gets the same number of records regardless of content.

4.2. 2.2 — Range partitioning (by region)

Change the loading phase: instead of round-robin, send each record to the worker that matches its `region` field. Spawn four workers named `eu`, `us`, `asia`, `africa`.

Hint: use `cluster.ask[LoadResult](region, ref => WorkerCommand.LoadData(...))` to route records by their region.

Run the pipeline. Observe that the shuffle phase now moves **zero** records for a region-based map — each key is already on the right worker.

4.3. 2.3 — Skewed range partition

Regenerate your data so that 70% of records have `region = us`:

```
def skewedRegion(): String =
  val r = Random.nextInt(100)
  if r < 70 then "us"
  else if r < 80 then "eu"
  else if r < 90 then "asia"
  else "africa"
```

Load with range partitioning. Run map + reduce. Which worker finishes last? How much time is wasted?

Discussion: this is **data skew**. The `us` worker processes 7× more records than the others. In Spark this shows up as a single task taking far longer than the rest — the "straggler" problem. Techniques to mitigate: salting keys, adaptive partitioning, broadcast joins.

4.4. 2.4 — Hash partitioning

Keep the skewed data but load with `hash(sale_id) % numWorkers` instead of by region. Records are distributed evenly regardless of region.

Compare map and reduce times with 2.3. Is the skew gone?

Discussion: hash partitioning destroys locality (EU records are spread across all workers) but guarantees even distribution. Range partitioning preserves locality but risks hot spots.

5. Part 3 — The shuffle

5.1. 3.1 — Measure shuffle cost

In the provided pipeline, print the number of records moved during the shuffle phase (sum of all values sent to all workers). With round-robin loading and a region-keyed map, all 10,000 records move. With range partitioning, zero move.

5.2. 3.2 — Repartition (shuffle as a tool)

Start with the skewed range partition from 2.3. After loading, **repartition** the data: read all records from all workers (via map), then redistribute by hash.

This is the distributed equivalent of a **shuffle**. Measure its cost, then run map + reduce on the repartitioned data. Is it faster than processing the skewed partition directly?

5.3. 3.3 — Scale up

Increase to 100,000 records and 8 workers. Run the pipeline with round-robin, range, and hash partitioning. Record timings.

Discussion: the shuffle moves (O(N)) records across the network. With simulated latency, this is the dominant cost. In real systems, the bottleneck is network bandwidth (shuffling terabytes of data). MapReduce frameworks optimize by combining on the map side, compressing intermediate data, and sorting to enable merge-based reduce.

6. Part 4 — Discussion

1. *Range vs.*
   hash partitioning: when would you prefer each? Think about queries that filter by region vs. queries that aggregate globally.
2. *What does a real MapReduce framework add?* Compare your implementation to Hadoop/Spark: fault tolerance (task retry), data locality (move compute to data), shuffle optimization, speculative execution for stragglers.
3. *Why is the shuffle so expensive?* Every record potentially crosses the network. In your cluster, each `ask` has 50–500ms latency. In real systems, a shuffle can move terabytes — the bottleneck shifts from latency to bandwidth.
4. *How does partitioning relate to replication?* In Lab 1.3.1 you replicated data for availability. Here you partition data for parallelism. Real systems combine both: Kafka partitions across brokers *and* replicates each partition for fault tolerance.

7. Bonus — Real dataset

Pick one of the following datasets, download a small slice, and run the full pipeline on it. Compare range vs.
hash partitioning and observe whether the data has natural skew.

7.1. Option A — GH Archive

Download one hour of GitHub events from gharchive.org (a single `.json.gz` file, ~50 MB decompressed). Each line is a JSON object with fields `type`, `repo.name`, `actor.login`, etc.

- **Range partition** by `type` (e.g. `PushEvent`, `WatchEvent`, …). `PushEvent` accounts for ~70% of events — natural skew.
- **Map:** extract `(repo_name, "1")` per event. **Reduce:** count events per repo.
- Compare with hash partition by event id.

7.2. Option B — NYC Taxi Trips

Download one month of Yellow Taxi trips from NYC TLC (CSV, ~100 MB per month).

- **Range partition** by pickup borough (Manhattan has ~70% of all trips — natural skew).
- **Map:** extract `(pickup_zone, fare_amount)`. **Reduce:** compute average fare per zone.
- Compare with hash partition by trip id.

7.3. Option C — Citi Bike

Download one month of trip data from Citi Bike (CSV, ~20 MB per month).

- **Range partition** by start station. A handful of Midtown stations have 10× more trips than outer-borough stations — natural skew.
- **Map:** extract `(station_name, trip_duration)`. **Reduce:** compute average trip duration per station.
- Compare with hash partition by trip id.

8. Key Takeaways

1. **Partitioning enables parallelism** — splitting data across workers lets you process shards concurrently.
2. **Hash partitioning spreads load evenly** — range partitioning preserves locality but risks hot spots.
3. **The shuffle is the most expensive operation** — moving data across the network dominates runtime. Minimizing shuffles is the primary optimization in Spark and Flink.
4. **Data skew kills performance** — one overloaded worker becomes the bottleneck while others idle.
5. **Map-reduce is simple in concept, hard in practice** — the framework overhead (shuffle, fault tolerance, scheduling) is where the real complexity lives.