**Limitations of MapReduce**

**Disk I/O heavy**

* Between map and reduce, data is written to disk (HDFS) → lots of reads/writes → slow performance.
* Intermediate data is materialized on disk → no in-memory processing → high latency.

This is the main reason Spark (in-memory) was created to outperform MapReduce.

**High latency / batch-only**

* MapReduce is optimized for batch processing, not for low-latency or interactive jobs.
* Cannot handle real-time streaming or interactive analytics efficiently.

**Rigid programming model**

* Only supports map and reduce primitives → can’t express complex processing (e.g. iterative machine learning, graph algorithms) naturally.
* Harder to chain operations without writing intermediate data to disk between jobs.

**Poor support for iterative algorithms**

* ML and graph algorithms often need to repeatedly process data (e.g. PageRank, KMeans).
* In MapReduce, every iteration is a separate job → new MapReduce job → disk I/O every time → very inefficient.

**Difficult error recovery for long chains**

* A failure late in a chain of MapReduce jobs often requires rerunning the entire chain or complex orchestration.

**Limited API expressiveness**

* Writing jobs requires a lot of boilerplate Java code (or Pig, Hive as wrappers).
* Harder to write clean, modular, reusable code compared to higher-level APIs (like Spark, Flink, or Beam).

**No built-in caching**

* MapReduce doesn't cache data between jobs → even static data is re-read and re-processed repeatedly.
* This increases latency and resource usage.

**Shuffle inefficiency**

* MapReduce’s shuffle phase is heavyweight → data is sorted, written to disk, copied over network → slow, especially for joins or group-bys.