PySpark File Read Modes

In **PySpark**, when reading files (like CSV, JSON, Parquet, etc.), you often use different **modes** to control how bad records or malformed data are handled.

Here are the **commonly used modes** when reading files in PySpark:

🔷 1. PERMISSIVE (default)

- **Behavior**: Corrupt records are set to **`null`** in all columns, and the whole bad record is stored in a special column **`_corrupt_record`**.

```
df = spark.read.option("mode", "PERMISSIVE") \
    .option("columnNameOfCorruptRecord", "_corrupt_record") \
    .json("path/to/file.json")
```

🔷 2. DROPMALFORMED

- **Behavior**: Drops rows that have malformed records.

```
df = spark.read.option("mode", "DROPMALFORMED") \
    .json("path/to/file.json")
```

🔷 3. FAILFAST

- **Behavior**: Throws an error as soon as it encounters a malformed record.

```
df = spark.read.option("mode", "FAILFAST") \
    .json("path/to/file.json")
```

🔷 Example for CSV with different modes

CSV with bad record:

`sample.csv`

```
id,name,age
1,John,30
2,Alice,twenty
3,Bob,25
```

---

1. PERMISSIVE

```
df = spark.read.option("header", "true") \
    .option("mode", "PERMISSIVE") \
    .option("inferSchema", "true") \
    .csv("sample.csv")

df.show()
```

🟢 Output:

```
+---+-----+----+
|id |name |age |
+---+-----+----+
|1  |John |30  |
```

```
|2  |Alice|null|
|3  |Bob  |25  |
+---+-----+----+
```

---

2. DROPMALFORMED

```
df = spark.read.option("header", "true") \
    .option("mode", "DROPMALFORMED") \
    .option("inferSchema", "true") \
    .csv("sample.csv")

df.show()
```

🟡 Output:

```
+---+----+---+
|id |name|age|
+---+----+---+
|1  |John|30 |
|3  |Bob |25 |
+---+----+---+
```

---

3. FAILFAST

```
df = spark.read.option("header", "true") \
    .option("mode", "FAILFAST") \
    .option("inferSchema", "true") \
    .csv("sample.csv")
```

🔴 Output:

```
Exception: java.lang.NumberFormatException: For input string: "twenty"
```

Would you like me to show similar examples for JSON/Parquet too?

To process **1 TB of data** efficiently in Spark using an HDFS-backed cluster with the below specs:

- **5 nodes**
- **Each node:** 8 cores, 32 GB RAM
- Target: **optimize Spark job and maximize HDFS throughput**
- Suggestion: **Use 5 cores per executor**

✅ Step-by-Step Spark Tuning Strategy:

🔷 1. Cores per Executor

- You said: use **5 cores per executor** ✅
- Why?
  - Avoids running too many tasks per executor, which causes GC overhead.
  - Leaves 1–2 cores for OS and HDFS daemons.
- Each node has 8 cores → Use **1 executor per node** with **5 cores**

✅ Set:

```
--executor-cores 5
```

🔷 2. Executors per Node

- With 8 cores per node:
  - If we assign 5 cores/executor → **1 executor per node**
  - Leaves 3 cores for OS, HDFS, YARN, etc. → Good for throughput

✅ Set:

```
--num-executors 5  # 1 executor per node × 5 nodes
```

🔷 3. Executor Memory

Each node has **32 GB** memory:

Reserve Memory:

- OS & Hadoop Daemons: ~2 GB
- YARN overhead: ~7–10% (default) → **`spark.yarn.executor.memoryOverhead`**

So, safely use ~27–28 GB for Spark executor

With 1 executor per node → assign ~25–28 GB

✅ Set:

```
--executor-memory 25g
```

Optionally:

```
--conf spark.yarn.executor.memoryOverhead=2g
```

🔷 4. Parallelism

- Rule of thumb: **2–3 tasks per core**
- Total cores = 5 nodes × 5 cores = **25 cores**
- So, target `spark.sql.shuffle.partitions` ≈ 50–100

✅ Set:

```
--conf spark.sql.shuffle.partitions=75
```

Also:

```
--conf spark.default.parallelism=75
```

🔷 5. HDFS Throughput Tips

To improve HDFS read/write:

- Use **compressed formats**: Parquet/Snappy
- Enable **vectorized reads**:

```
--conf spark.sql.parquet.enableVectorizedReader=true
```

- Leverage **coalesce/repartition** before writing if small files issue
- Avoid **too many small files** (use `.coalesce()` or `.repartition()` wisely)

✅ Final Spark Submit Example

```
spark-submit \
  --executor-cores 5 \
  --num-executors 5 \
  --executor-memory 25g \
  --conf spark.sql.shuffle.partitions=75 \
  --conf spark.default.parallelism=75 \
  --conf spark.yarn.executor.memoryOverhead=2g \
  your_script.py
```

Would you like me to include **autoscaling**, **dynamic allocation**, or **EMR-specific tips** too?

To process **5 TB of data** on a **50-node cluster**, each with **16 cores and 128 GB RAM**, you'll want to **optimize Spark resource allocation** for maximum **parallelism, memory efficiency, and HDFS throughput**.

✅ 1. Cluster Overview

Attribute	Value
Nodes	50
Cores per node	16
Total cores	800 (50 × 16)
Memory per node	128 GB
Total memory	6.25 TB (128 × 50)
Total Data	5 TB

✅ 2. Key Goals

- Efficient **parallelism** and **task distribution**
- Avoid **GC overhead** (not too many cores per executor)
- Leave headroom for **OS and Hadoop daemons**
- Maximize **HDFS read/write throughput**
- Prevent **OOM** (Out of Memory) errors and **data skew**

✅ 3. Resource Tuning Strategy

🔷 A. Executor Cores

- Recommended: **5 cores per executor**

- Why?
  - Avoid large thread pools → better GC performance
  - HDFS performs best with ~5 tasks per node for disk/network

✅ Set:

```
--executor-cores 5
```

🔷 B. Executors per Node

Each node: 16 cores, 128 GB RAM
→ 3 executors per node is ideal:

- Each executor → 5 cores
- Total used cores = 3 × 5 = 15 (1 left for OS)
- Memory split: 128 / 3 ≈ 42 GB per executor

✅ Set:

```
--num-executors 150   # 3 executors/node × 50 nodes
```

🔷 C. Executor Memory

- Total memory per executor: ~42 GB
- Subtract memory overhead (YARN default: 7% or set manually)

E.g., assign:

```
--executor-memory 38g
--conf spark.yarn.executor.memoryOverhead=4g
```

🔷 D. Parallelism

- Total executor cores = 150 executors × 5 cores = 750
- Recommended:
  - `spark.default.parallelism` = 2–3 × total cores → ~1500–2250
  - `spark.sql.shuffle.partitions` = ~1500–2000

✅ Set:

```
--conf spark.default.parallelism=2000
--conf spark.sql.shuffle.partitions=2000
```

✅ 4. Storage Format & Compression

- Prefer **Parquet** + **Snappy**: columnar & compressed
- Enables **predicate pushdown**, **vectorized reads**

✅ Set:

```
--conf spark.sql.parquet.enableVectorizedReader=true
```

✅ 5. HDFS Considerations

- Avoid **small files**: repartition before write
- Coalesce or Repartition output if needed:

```
df.repartition(2000).write.parquet(...)
```

✅ 6. Other Config Recommendations

- **Dynamic allocation** if jobs vary:

```
--conf spark.dynamicAllocation.enabled=true
--conf spark.dynamicAllocation.minExecutors=50
--conf spark.dynamicAllocation.maxExecutors=150
```

- GC tuning (optional for heavy workloads):

```
--conf spark.executor.extraJavaOptions="-XX:+UseG1GC"
```

✅ Final Spark Submit Example:

```
spark-submit \
  --executor-cores 5 \
  --executor-memory 38g \
  --num-executors 150 \
  --conf spark.yarn.executor.memoryOverhead=4g \
  --conf spark.default.parallelism=2000 \
  --conf spark.sql.shuffle.partitions=2000 \
```

```
  --conf spark.sql.parquet.enableVectorizedReader=true \
  your_script.py
```

🧠 Factors to Monitor During Execution

Category	Metrics / Symptoms
CPU	Core underutilization → increase cores
Memory	OOM or GC overhead → reduce cores/executor
Shuffle	Skew or spill → repartition, adjust partitions
I/O	Slow HDFS reads → check network or I/O
Small files	Too many → coalesce before write

Would you like help with a **resource planning table**, or **autoscaling setup** for dynamic workloads like streaming or interactive jobs?

Great question — let's clarify where the **5 TB** data size factors into the Spark configuration.

🔷 Where Is the 5 TB Considered?

The **5 TB** data size **influences** the following **tuning decisions**, even though it's not directly set as a Spark parameter:

✅ 1. Parallelism Planning

- The **amount of data** you process **drives how many tasks** Spark should generate.
- **Rule of thumb**:
  - **1 Spark task per 128 MB to 256 MB** of input data.
  - For 5 TB = 5120 GB = 5,242,880 MB
    → Target: ~**20,000 to 40,000 tasks**

So, based on this, we derived:

```
--conf spark.default.parallelism=2000
--conf spark.sql.shuffle.partitions=2000
``> (This assumes Spark splits & stages will break up the job automatically
across stages.)

But if needed, you can do:
```python
df.repartition(5000)   # to explicitly increase parallelism before writing
```

✅ 2. Executor Count & Sizing

- We expect each executor to process a **chunk of the 5 TB** (say ~35–40 GB per executor).
- With **150 executors**, each handles about:
```

$$\frac{5\,\text{TB}}{150} = \frac{5120\,\text{GB}}{150} \approx 34\,\text{GB per executor}$$

That justifies assigning ~**38 GB memory per executor** with 5 cores.

✅ 3. Shuffle Partition Count

- Shuffle-intensive stages (like joins, aggregations) will spill if partitions are too large.
- Keeping **~128–256 MB per shuffle partition** helps avoid spill.

With 5 TB of data:

$$\frac{5242880\,\text{MB}}{256\,\text{MB}} = 20,480 \text{ partitions (max)}$$

So, setting `` `spark.sql.shuffle.partitions = 2000–4000` `` is a **balanced default** unless the job is very skewed.

🔁 Summary: Where 5 TB is *used implicitly*

Decision Area	How 5 TB impacts it
`` `spark.sql.shuffle.partitions` ``	More data → more partitions to avoid spills
Executor memory	Data per executor must fit in memory
Parallelism	More tasks → faster throughput if cluster can handle
Data formats	Compression (Parquet/Snappy) reduces data load in memory
I/O configuration	Need enough cores and memory to handle HDFS throughput

Would you like to calculate **estimated job duration** or **HDFS block strategy** for this data too?

Great question! These **three configurations** — **`repartition()`**, **`spark.sql.shuffle.partitions`**, and **`spark.default.parallelism`** — are all related to **parallelism and shuffling in Spark**, but they apply in **different contexts**.

🔷 1. `repartition()`

✅ What it is:

- A **transformation** in Spark that **reshuffles data** into a given number of partitions **at runtime**.
- It causes a **full shuffle** of data across the cluster.

✅ When to use:

- When you **explicitly want to change the number of partitions**.
- Useful before **expensive operations** (like joins, groupBy, writes) to avoid skew or small files.

✅ Example:

```
df.repartition(200)  # creates 200 partitions (i.e., 200 parallel tasks)
```

🔷 2. `spark.sql.shuffle.partitions`

✅ What it is:

- A **Spark configuration setting** that defines the **default number of partitions** created **after a shuffle** in **Spark SQL operations** (like `groupBy`, `join`, `distinct`, `orderBy`).
- Default is **200**.

✅ When it applies:

- Automatically used when a **shuffle happens in SQL/DataFrame APIs** (not RDDs).
- Controls how many output partitions are created **after** a shuffle.

✅ Example:

```
spark.conf.set("spark.sql.shuffle.partitions", 1000)
```

- This will make Spark **create 1000 output partitions** after any `groupBy`, `join`, etc.

🔷 3. `spark.default.parallelism`

✅ What it is:

- A **core Spark setting** that determines the **default number of partitions** for **RDD operations** like `parallelize()`, `reduceByKey()`, and `join()` (for RDDs).
- Default is usually **number of all cores** in the cluster.

✅ When it applies:

- Used for **RDD APIs**, or **DataFrame operations without a specified partition count** (like `df.rdd.reduceByKey()`).
- Also used as a **fallback** if no explicit partition number is set in some transformations.

✅ Example:

```
spark.conf.set("spark.default.parallelism", 1000)
```

🧠 Summary Table

Feature	Applies To	Triggers Shuffle	Manual/Auto	Typical Use Case
`repartition(n)`	DataFrame/RDD	✅ Yes	Manual	Set partitions before write/join/groupBy
`spark.sql.shuffle.partitions`	DataFrame (SQL APIs)	✅ Yes (post-shuffle)	Auto	Controls partitions after SQL shuffle
`spark.default.parallelism`	RDDs & fallback cases	❌ No (by itself)	Auto	Sets default parallelism for RDD jobs

---

🔧 Best Practices

- Use `repartition(n)` **only when necessary** (it causes full shuffle).
- Tune `spark.sql.shuffle.partitions` based on **cluster size and data volume** (e.g., 2–3x total cores).
- Set `spark.default.parallelism` when working with **RDDs**, or to override Spark's default.

💡 Bonus: Avoiding small files

If your output is generating too many small files:

```
df.coalesce(200).write.parquet("...")
```

(`coalesce()` minimizes shuffle unlike `repartition()`)

Would you like a flowchart or decision tree on when to use each of these?

In PySpark, when **writing DataFrames to storage** (like HDFS, S3, local disk, or a database), you can specify a **write mode** that determines **how existing data is handled**.

✅ Supported Write Modes in PySpark

Write Mode	Behavior
`append`	Adds data to the existing directory or table.
`overwrite`	Replaces existing data at the path or table.
`ignore`	Does nothing if data or path already exists.
`error` or `errorifexists` (default)	Fails if the path or table exists.

🔷 1. `append`

- **Adds** data to existing files/table without deleting anything.
- Commonly used in incremental pipelines.

```
df.write.mode("append").parquet("path/to/output")
```

🔷 2. `overwrite`

- **Deletes existing data** at the output path and writes new data.
- Be cautious: it fully replaces the existing data.

```
df.write.mode("overwrite").csv("path/to/output")
```

Optionally with partition overwrite mode:

```
spark.conf.set("spark.sql.sources.partitionOverwriteMode", "dynamic")
```

🔷 3. `ignore`

- If path/table already exists, **do nothing**.
- Does **not throw error** or overwrite anything.

```
df.write.mode("ignore").json("path/to/output")
```

🔷 4. `error` or `errorifexists` (default)

- Fails if the output path already exists.
- Useful to prevent accidental overwrite.

```
df.write.mode("error").parquet("path/to/output")
```

✅ Format Example

```
df.write \
  .format("parquet") \
  .mode("overwrite") \
  .save("/data/processed/output")
```

🧠 Note:

- These modes apply to **DataFrameWriter API**.
- Not all connectors (like JDBC) support all modes.

Would you like examples for **JDBC write modes**, or how `overwrite` behaves with **partitioned tables** in Hive/S3?

Here's a **detailed comparison** of **AWS Glue vs EMR vs EMR Serverless**, tailored for a **Data Engineering use case** like yours — with real-world examples, pros/cons, and when to choose which one.

✅ At a Glance: AWS Glue vs EMR vs EMR Serverless

Feature	AWS Glue	EMR (Classic)	EMR Serverless
Type	Serverless ETL Service	Managed Cluster for Big Data	Serverless Spark/Hive Engine
Compute Provisioning	Fully managed	You manage EC2 instances (manual or auto)	Auto-managed, on-demand compute
Use Case	ETL (Batch), Data Catalog	Complex pipelines, ML, Streaming	On-demand Spark jobs, ad-hoc processing
Start-up Time	~1-2 minutes	5–10 minutes (cluster spin-up)	<1 minute (cold start), ~sec (warm start)

Feature	AWS Glue	EMR (Classic)	EMR Serverless
Cost Model	Pay per second	Pay per EC2 instance-hour	Pay per second (vCPU + GB-hour)
Language Support	PySpark, Scala, Python (Glue version)	Spark (any), Hadoop, Hive, Presto, HBase	Spark (PySpark, Hive SQL)
Scaling	Auto-scaled	Manual or Auto Scaling	Auto-scaled (per job basis)
Tight S3 Integration	✅ Yes	✅ Yes	✅ Yes
Glue Catalog Integration	✅ Native	✅ Via Hive Metastore integration	✅ Native
Dev Experience	Visual Studio, Jupyter-style notebooks	SSH, Notebooks, Livy, EMR Studio	Jupyter notebooks, APIs, Glue jobs interface

---

🔍 Feature-by-Feature Breakdown

🔷 1. Provisioning & Startup Time

Feature	Glue	EMR	EMR Serverless
Start time	1–2 min	5–10 min	Cold start < 1 min
Scaling	Auto	Manual/Auto	Fully automatic
Idle Cost	None	You pay for idle EC2	None (no cluster)

🔷 2. Performance & Customization

Aspect	Glue	EMR	EMR Serverless
Performance Tuning	Limited	Full control (EC2 type, configs)	Some configs exposed
Spark Version Control	AWS managed (slightly older)	You control version	Limited but newer versions available
Resource Isolation	Shared in Glue workers	Full EC2 control	Per job runtime isolation

---

🔷 3. Supported Engines

Engine	Glue	EMR	EMR Serverless
PySpark	✅ Yes	✅ Yes	✅ Yes
Scala Spark	❌ No	✅ Yes	❌ (only PySpark)
Hive / SQL	✅ Yes	✅ Yes	✅ Yes
Presto / Trino	❌ No	✅ Yes	❌ No
Hadoop MapReduce	❌ No	✅ Yes	❌ No

🔷 4. Real-World Use Cases

Use Case	Best Tool	Example
Simple ETL (daily S3 → S3 with transformations)	**Glue**	Reading Parquet from S3, transform in PySpark, write to Redshift
Machine Learning + Feature Engineering	**EMR**	Run Spark MLlib, XGBoost pipelines at scale on EC2
Ad-hoc Querying or **Event-Driven Spark**	**EMR Serverless**	Trigger Spark SQL job on file arrival using Lambda + EMR Serverless
Streaming ETL with Kafka + Spark	**EMR**	Real-time log ingestion from Kafka + Spark Structured Streaming
100s of small, parallel jobs	**Glue or EMR Serverless**	Scalable batch processing (1 job per partition)

🧪 Examples

🟡 AWS Glue Example: PySpark ETL Job

```
from awsglue.context import GlueContext
glueContext = GlueContext(SparkContext.getOrCreate())

df = glueContext.create_dynamic_frame.from_catalog(
    database="mydb", table_name="s3_input_table"
).toDF()

transformed = df.filter(df["amount"] > 100)

transformed.write.mode("overwrite").parquet("s3://my-bucket/output/")
```

🔵 EMR (Classic) Example: Cluster Setup + PySpark Submit

```
aws emr create-cluster \
  --name "MySparkCluster" \
  --release-label emr-6.15.0 \
  --applications Name=Spark \
  --ec2-attributes KeyName=my-key \
  --instance-type m5.xlarge \
  --instance-count 3 \
  --use-default-roles \
  --log-uri s3://my-logs/emr/
```

Then:

```
spark-submit s3://my-scripts/my_etl.py
```

🟢 EMR Serverless Example: Submit Spark Job via CLI

```
aws emr-serverless start-job-run \
  --application-id <app-id> \
  --execution-role-arn arn:aws:iam::... \
  --job-driver '{
    "sparkSubmit": {
      "entryPoint": "s3://my-job/main.py"
    }
  }' \
  --configuration-overrides '{
    "monitoringConfiguration": {
      "s3MonitoringConfiguration": {
        "logUri": "s3://my-logs/"
      }
    }
  }'
```

🏁 When to Use What?

Scenario	Use
Fast setup, no infra ops	AWS Glue
Custom tuning, high performance	EMR
On-demand Spark jobs, event-driven	EMR Serverless
Streaming, Presto, Hadoop workloads	EMR
Short ETL jobs, cheap scale-out	Glue or EMR Serverless

💰 Cost Comparison (Approx.)

Service	Cost Control
AWS Glue	Per DPU-second (~$0.44/DPU/hour)
EMR Classic	Per EC2 + EBS + EMR charge
EMR Serverless	Per second (vCPU & memory usage)

✅ Recommendation Based on Use Case

Use Case	Best Choice
ETL automation with catalog	Glue
Data Science workloads	EMR (Jupyter + Spark)
Event-triggered Spark jobs	EMR Serverless
Heavy batch + tuning need	EMR (EC2 tuning)

Would you like a **decision tree diagram** or a **cost estimator Excel sheet** comparing these options for a specific use case (like 1 TB/day ETL)?