🔹 Q1. Compare Parquet, Avro, and ORC in terms of structure, performance, and use

Feature / Format	Parquet	Avro	ORC
Data Layout	Columnar	Row-based	Columnar
Compression	Good	Moderate	Best-in-class
Predicate Pushdown	✅ Yes	❌ No	✅ Yes
Splittable	✅ Yes	✅ Yes	✅ Yes
Schema Evolution	✅ Supported, but limited flexibility	✅ Best-in-class	✅ Supported, less flexible than Avro
Read Efficiency	✅ Great for column-based queries	❌ Slower for column queries	✅ Excellent for Hive workloads
Write Efficiency	Moderate	✅ Fast	Moderate
Support in Spark	✅ Native support	✅ Native support	✅ Supported (but better in Hive setups)
Best Use Cases	Analytics, Data Lakes, Spark SQL	Kafka pipelines, CDC, serialization	Hive warehouses, Hadoop-native systems

✅ Why Avro (Row-based) Doesn't Support Predicate Pushdown:

- **Avro is a row-based format**, meaning it stores entire rows together (all columns of one record).
- To filter on a specific column (e.g., `WHERE age > 30`), Spark must **read the full row** first — it cannot skip reading other columns efficiently.

🔍 Predicate Pushdown Needs Columnar Layout

Formats like **Parquet** and **ORC** are **columnar**, which means:

- Each column is stored separately.
- So Spark can **read only the needed column** (e.g., `age`) and **skip the rest**.
- This enables **predicate pushdown**: filters are pushed down to the storage layer → less I/O → faster queries.

IMP : SPARK STORES DATA AS AN CATALYST DATAFRAME, NOT AS AVRO/ PARQUET

Stage	Format Used	Notes
Kafka → Spark	**Avro → Catalyst DF**	Avro decoded into Catalyst memory
Computation in Spark	**Catalyst (RAM)**	No Parquet here
Writing to Delta Lake / S3 / HDFS	**Parquet**	Controlled by sink format

What	Format
Input from Kafka	Avro (or JSON, CSV etc.)
Spark computation	Catalyst DF (in-memory)
Spill to disk	Internal binary (not Parquet/Avro)
Sink (e.g., Delta)	Parquet + _delta_log

Stage	Stored as	Push-down possible?
Kafka Avro ➔ Spark (ingestion)	Avro bytes → Catalyst rows	**No**
Parquet ➔ Spark (read)	Parquet → Catalyst rows	**Yes**, before rows hit memory

Predicate push-down is a *read-time* optimisation.

It works only if the source format advertises and supports it (Parquet/ORC/JDBC).

Once data is inside the Catalyst engine , Spark just filters in RAM—format no longer matters. **(i.e if it is Parquet - Catalyst DF knows it supports predictive pushdown, for further join/, etc operations , PP is supported, else Not)**

Spark knows during read time if pushdown is possible.(i.e from avro/Parquet/CSV/JSON just before getting converted to catalyst DF)

If source = Avro → pushdown is not possible → everything comes into memory.

🔹 Q2. What is predicate pushdown? Which formats support it?

✅ Concept:

- Lets Spark **skip unnecessary blocks** of data at the file reader level
- **Parquet/ORC** support it due to columnar structure
- **Avro, JSON, CSV** — no real predicate pushdown

🧠 Example:
python
CopyEdit
```
df.filter("region = 'US'")
```

Only the 'region' column is read from disk in **Parquet.**

🔹 Q3. What is the small file problem in Spark? How do you fix it?

Issue	Impact
Too many small files (e.g., 100k files <1MB)	File lookup overhead, task explosion, poor parallelism

✅ **Fixes**:

- Write fewer, larger files using `.repartition(n)(When shuffling)` or `.coalesce(n) (no shuffling)`
- Use `spark.sql.files.maxPartitionBytes` config
- Use Delta's `OPTIMIZE` or compaction jobs

🧠 Real Scenario:

> Your streaming job writes 1 file per trigger → leads to 10,000+ small files on S3. Fix by batching and compacting downstream.

🔹 **Q4. Explain `inferSchema`, `mergeSchema`, and their performance impact.**

Option	Role	Impact
`inferSchema`	Guess column types from data	Costly for large datasets
`mergeSchema`	Combine schemas across files (Parquet)	Useful for evolution, but **very expensive** on large datasets

✅ **Best Practice**:

- Always define schemas explicitly in production
- Avoid `mergeSchema = true` in mainline ingestion jobs

🔍 **Explanation per file format:**

File Format	Schema Inference Needed?	`inferSchema` Used?	Notes
CSV	✅ Yes (no schema stored)	✅ Yes (optional)	Must infer or define manually
JSON	✅ Yes	✅ Yes (optional)	Can infer, but expensive
Avro	❌ No (schema stored in header)	❌ Not needed	Schema is embedded in file
Parquet	❌ No (schema stored in footer)	❌ Ignored	Schema is read directly
ORC	❌ No	❌ Not needed	Like Parquet, optimized for analytics

Avro/ Paquet/ ORC - **file formats** that stores schema **internally** in the file footer.

🔹 Q5. How do you handle corrupted/bad records in ingestion?

🔹 **What Are Bad Records?**

Rows that:

- Have **incomplete data** (e.g., missing values for required columns)
- Have **type mismatches** (e.g., string in an integer column)
- Are **malformed** (e.g., bad JSON format)

✅ Available Options in `.option("mode", ...)`

Mode	Behavior
`"PERMISSIVE"`	*(Default)* Tries to parse rows. If bad, puts `null` in corrupt columns and logs in `_corrupt_record`.
`"DROPMALFORMED"`	Silently **drops bad rows.** No trace.
`"FAILFAST"`	**Fails immediately** on first corrupt record.

📌 Example: Reading Malformed JSON

File: `data.json`

```json
{"id": 1, "name": "Alice"}
{"id": 2, "name": "Bob"}
{"id": 3, "name": "Charlie"
{"id": 4, "name": "David"}
```

Row 3 is missing a closing brace.

Code:

```python
df = spark.read \
    .option("mode", "PERMISSIVE") \
    .option("columnNameOfCorruptRecord", "_corrupt_record") \
    .json("data.json")

df.show(truncate=False)
```

Output:

```pgsql
+----+-------+--------------------------+
| id | name  | _corrupt_record          |
+----+-------+--------------------------+
| 1  | Alice | null                     |
| 2  | Bob   | null                     |
|null| null  | {"id": 3, "name": "Charlie" |
| 4  | David | null                     |
+----+-------+--------------------------+
```

🔑 Takeaway:

- **Never silently drop** malformed data in production.
- Always **log, isolate, or quarantine** bad records.
- Use `"PERMISSIVE"` with `_corrupt_record` and route them to **Delta Lake quarantine tables**.

🔹 Q6. What are common ingestion pipeline optimizations?

Area	Optimization
Partitioning	Use `partitionBy("dt")` when writing
File size	Aim for 100–500 MB per file
Schema	Provide `.schema()` instead of inferring
Input splits	Avoid deep nested folders unless needed
Compression	Use `snappy` (default for Parquet) for fast compression/decompression

---

🔹 Q7. What's the difference between `.save()`, `.saveAsTable()`, and `.insertInto()`?

Method	Use When	Behavior
`.save()`	Write to path	Just writes files
`.saveAsTable()`	Register DataFrame as Hive table	Creates table + writes data

`.insertInto()`	Insert into existing table	Table must exist, schema must match exactly

---

🔹 Q8. What happens internally when Spark reads a file?

🔷 Step-by-Step Internal Workflow

Step	What Happens	Example
1️⃣	**Spark contacts the file system** (like S3, HDFS, DBFS)	When you do `spark.read.parquet("s3://bucket/path")`, Spark queries S3 to get metadata (file list, sizes, locations).
2️⃣	**File is split into logical partitions**	A 1 GB Parquet file might be split into 8 partitions (~128 MB each by default) based on `spark.sql.files.maxPartitionBytes`.
3️⃣	**Schema is read (or inferred)**	- For Parquet: Schema is read from the file footer (fast). - For CSV/JSON: Schema might be **inferred** by reading sample rows (slow).
4️⃣	**Executors read file blocks**	Each partition is sent to a task, which runs on an executor. That task **reads the corresponding file split (block)**.
5️⃣	**Deserialize → Catalyst DataFrame**	Data is deserialized into Spark's internal Catalyst format (binary, optimized).
6️⃣	**Lazy evaluation**	No actual reading/transformation happens yet — Spark just builds the logical plan. **Actual work happens only when you call an action** like `.show()`, `.collect()`, `.write()`.

🔹 Q9. What's a good ingestion-to-storage pattern?

✅ Stage-wise Explanation

Layer	Format Used	What It Stores	Why It's Needed
Bronze	Avro	Raw Kafka data (possibly nested, unclean)	Keeps original source; Avro supports **schema evolution** well
Silver	Parquet	Flattened, cleaned, enriched data	Columnar format → fast analytical reads, partitioning works well
Gold	Delta	Aggregated, optimized data for consumption	Supports **time travel, MERGE, exactly-once**, BI dashboards