12 DataFrame Writer API and Partition Pruning

What is DataFrame Writer API

The **DataFrame Writer API** is used to **persist a DataFrame to storage**.

In PySpark, it is accessed via:

```
df.write
```

It supports writing data to:

- Files (HDFS, S3, local)
- Tables (Hive metastore, Spark catalog)
- Different formats (csv, parquet, json, delta, etc.)

Basic write syntax

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

  or

df.write \
  .format("parquet") \
  .mode("overwrite") \
  .option("path", "/data/output") \
  .save()

spark.read.format("csv").options
spark.read.csv
```

Equivalent shorthand:

```
df.write.mode("overwrite").parquet("/data/output/")
```

Write modes

Mode	Behavior
`error` (default)	Fails if path exists
`overwrite`	Deletes existing data
`append`	Adds new data
`ignore`	Skips write if path exists

Example:

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

Common file formats

Parquet (recommended default)

- Columnar
- Compressed
- Schema stored with data
- Fast for analytics

```
df.write.parquet("/data/orders")
```

CSV

- Text based
- No schema stored
- Needs options

```
df.write \
  .option("header", "true") \
```

```
  .csv("/data/orders_csv")
```

JSON

- Semi structured
- Schema inferred on read

```
df.write.json("/data/orders_json")
```

ORC

- Columnar
- Common in Hive ecosystems

```
df.write.orc("/data/orders_orc")
```

Partitioning while writing

Partitioning physically organizes data into folders.

```
df.write \
  .partitionBy("order_date") \
  .parquet("/data/orders")
```

Folder structure:

```
order_date=2025-01-01/
order_date=2025-01-02/
```

Demo

```
orders_schema = 'order_id int , customer_id int ,product_id int, amount dou
ble , order_date string, order_status string , state string'
orders_df = spark.read.csv("/data/orders_500mb.csv" , schema = orders_s
chema , header = True)
orders_df.write.format("csv").mode("overwrite").partitionBy("order_statu
```

```
s").option("header","true").save("/data/writer_demo1/")
orders_df.write.format("csv").mode("overwrite").partitionBy("order_statu
s","state").option("header","true").save("/data/writer_demo1/")

df =  spark.read.option("header","true").schema(orders_schema).csv("/dat
a/writer_demo1/")
df.filter("state = 'Goa' and order_status=='DELIVERED' ").write.format("cs
v").mode("overwrite").option("header","true").save("/data/writer_demo_filt
er/")
```

Important rules

- Partition columns are **not written inside files**
- Partitioning should be done on columns which do not have very high distinct values like order_id.
- Over partitioning causes small files

Bucketing (tables only)

Bucketing distributes data into fixed number of buckets.

```
df.write \
  .bucketBy(8, "product_id") \
  .sortBy("product_id") \
  .saveAsTable("orders_bucketed")

  #demo
orders_schema = 'order_id int , customer_id int ,product_id int, amount dou
ble , order_date string, order_status string , state string'
orders_df = spark.read.csv("/data/orders_500mb.csv" , schema = orders_s
chema , header = True)
orders_df.write.format("csv").mode("overwrite").bucketBy(4,"customer_i
d").saveAsTable("writer_demo1")
```

Notes

- Works only with `saveAsTable`

- Requires Hive support
- Bucket count is fixed
- it helps in optimizing joins

Writing to tables

saveAsTable

Creates or overwrites a table.

```
df.write \
  .mode("overwrite") \
  .saveAsTable("orders")

df_orders.write.format("parquet").mode("overwrite").partitionBy("order_sta
tus").saveAsTable("orders")
```

- Data + metadata stored
- Managed or external table

Schema handling during write

- Schema is written automatically for columnar formats
- CSV does not store schema

Options commonly used

```
df.write \
  .option("compression", "snappy") \
  .option("maxRecordsPerFile", 1000000) \
  .parquet("/data/output")
```

Common options:

- compression

- header
- delimiter
- maxRecordsPerFile

Writing with repartition vs coalesce

```
df.repartition(10).write.parquet("/data/output")
df.coalesce(5).write.parquet("/data/output")
```

- Number of output files = number of partitions
- Use coalesce to reduce files
- Use repartition to rebalance data

Noop writer (used for execution only)

```
df.write.format("noop").save()
```

- Triggers full execution
- No data written
- Used for debugging and explain plans

Performance tips

- Prefer Parquet or ORC
- Control number of output files
- Avoid too many partitions
- Partition only on low cardinality columns
- Use bucketing for frequent joins