Contents

[Follow-up Questions 4](#_Toc202539787)

[Q (ChatGPT). Benefits of using AVRO schema 4](#_Toc202539788)

[Answer 4](#_Toc202539789)

[Q (ChatGPT). AVRO & Parquet ? 6](#_Toc202539790)

[Answer 7](#_Toc202539791)

[Q (ChatGPT) – Avro & Parquet (2) 8](#_Toc202539792)

[Answer 9](#_Toc202539793)

[Q (ChatGPT) – JSON 11](#_Toc202539794)

[Answer 11](#_Toc202539795)

[Q. (ChatGPT) “Controller service” VS. “Processor “ in NiFi 13](#_Toc202539796)

[Answer 13](#_Toc202539797)

[Q (ChatGPT) – Tale/Narrative Fashion 15](#_Toc202539798)

[Q. (ChatGPT) – HDFS ? 20](#_Toc202539799)

[Answer 20](#_Toc202539800)

Great! Let's walk through each of the mentioned NiFi processors and controller services, connecting **what they do**, **how they do it**, and **how they fit together in a real-world flow** — using your Netflix data example.

**✅ 1. UpdateAttribute Processor**

**What it does:**  
Adds or updates attributes (metadata) on FlowFiles.

**How it performs this task:**  
You configure it with key-value pairs. For example, you add:

* schema.name = NetflixTitleSchema

This means every FlowFile that passes through this processor will have an attribute called schema.name set to NetflixTitleSchema.

**Why it matters / Real-world example:**  
NiFi processors like PartitionRecord or JsonRecordSetWriter can reference this schema.name attribute to dynamically decide which schema to use when reading or writing records.

**Simple analogy:**  
Think of a FlowFile as a package, and attributes are sticky notes on the package. You’re adding a sticky note that says, "Use NetflixTitleSchema when you unpack this!"

**✅ 2. CSVReader Controller Service**

**What it does:**  
Reads and parses CSV-formatted content from incoming FlowFiles and converts them into a record-oriented structure.

**How it performs this task:**  
You configure it to:

* **Treat First Line as Header = true** (this tells it that the first row contains column names like show\_id, title, etc.)

It uses those headers to map each row to a record.

**Real-world example:**  
Let’s say you receive this CSV:

show\_id,type,title

s1,Movie,The Irishman

s2,TV Show,Stranger Things

The CSVReader will turn this into structured records:

{"show\_id": "s1", "type": "Movie", "title": "The Irishman"}

**Why it’s important:**  
Downstream processors like PartitionRecord work with structured data (records), not raw text or CSV. So, this reader is how we convert raw CSV into something NiFi can intelligently process.

**✅ 3. AvroSchemaRegistry Controller Service**

**What it does:**  
Stores named Avro schemas centrally, so other processors (like readers or writers) can reference and reuse them.

**How it performs this task:**  
You define a new schema with a name (e.g., NetflixTitleSchema) and paste in your Avro schema.

Example:

{

"name": "NetflixTitleSchema",

"type": "record",

"fields": [

{"name": "show\_id", "type": "string"},

{"name": "type", "type": ["null", "string"], "default": null},

{"name": "title", "type": ["null", "string"], "default": null}

]

}

**Why it’s useful:**  
Now other processors (like JsonRecordSetWriter) can just say “use NetflixTitleSchema” without repeating the schema definition.

**Analogy:**  
It’s like a library where schemas are books. Instead of copying the whole book each time, processors can borrow it by name.

**✅ 4. JsonRecordSetWriter Controller Service**

**What it does:**  
Writes the output records (coming from another processor) as JSON format.

**How it performs this task:**  
You configure it to:

* **Schema Access Strategy:** "Use 'schema.name' Attribute"
* **Schema Registry:** AvroSchemaRegistry
* This means: "Look at the attribute on the FlowFile (schema.name) and fetch that schema from the Avro Schema Registry."

**Practical example:**  
Using the earlier records from the CSV:

{"show\_id": "s1", "type": "Movie", "title": "The Irishman"}

This gets written as valid JSON, according to the NetflixTitleSchema.

**✅ 5. PartitionRecord Processor**

**What it does:**  
Splits one large flow of records into multiple smaller flows based on a field’s value.

**How it performs this task:**  
You configure it with:

* **Partitioning Field:** type (e.g., "Movie" or "TV Show")
* **Record Reader:** CSVReader
* **Record Writer:** JsonRecordSetWriter

So, when you send in the CSV records, this processor:

1. Reads all records with the CSVReader
2. Groups them by the value in type field
3. Writes separate FlowFiles with only "Movie" or only "TV Show" records.

**Example:**  
Input:

s1,Movie,The Irishman

s2,TV Show,Stranger Things

s3,Movie,Inception

Output:

* One FlowFile with all Movie entries
* Another with all TV Show entries

**Why it’s powerful:**  
Now you can route each FlowFile separately — maybe movies go to one database and shows to another.

**✅ Summary Flow (Putting It All Together)**

| **Step** | **What Happens** | **Tool/Processor** |
| --- | --- | --- |
| 1 | FlowFile gets schema.name = NetflixTitleSchema | UpdateAttribute |
| 2 | CSV gets parsed into structured records | CSVReader |
| 3 | Records are grouped by content type (Movie/TV Show) | PartitionRecord |
| 4 | JSON output is written using Avro schema | JsonRecordSetWriter |
| 5 | Avro schema is managed in one place | AvroSchemaRegistry |

Let me know if you'd like a visual diagram, YAML snippets, or a sample template for these NiFi components.

# Follow-up Questions

## Q (ChatGPT). Benefits of using AVRO schema

why the step "set up an AvroSchemaRegistry (controller service)" was required in "2. Prepare the Data (Parse, Structure, and Partition)" ? Wasn't it possible that CSV input file be directly used to get the JSON file

## Answer

Excellent question! Let's break this down clearly.

**❓ Why set up an AvroSchemaRegistry controller service in Step 2?**

You're absolutely right that **CSV can be directly converted to JSON**. However, setting up an **AvroSchemaRegistry** brings **key advantages** and **specific purposes** in this workflow.

**✅ Yes, it *is* possible to convert CSV directly to JSON without using AvroSchemaRegistry**

* You can use:
  + CSVReader (to parse the CSV)
  + and JsonRecordSetWriter (to output JSON)
* This works **without** needing a schema registry, **as long as**:
  + The JSON writer is configured to **infer the schema**, or
  + You use a hardcoded schema inside the processor.

BUT…

**🧠 So why use AvroSchemaRegistry anyway? What’s the benefit?**

Using the **AvroSchemaRegistry** provides the following **important benefits**:

**1. Centralized Schema Management**

* Define your data structure **once**, in one place.
* Avoid repeating schema definitions in multiple processors.
* Especially useful when many processors or flows share the same schema.

**2. Strong Data Validation**

* The schema **defines the data types** (e.g., release\_year as int, cast as nullable string, etc.)
* If the CSV has **bad or unexpected data**, NiFi can **catch errors early** during parsing.

**3. Schema Evolution Support**

* If the input data changes (e.g., a new column is added later), you can **update the schema once** in the registry.
* All processors referencing that schema **automatically adapt**.

**4. Cleaner and Controlled JSON Output**

* Without a schema, the JSON writer **infers types**, which might be:
  + inaccurate (e.g., treating numbers as strings), or
  + inconsistent (if column data varies across rows).
* With a schema, you get **consistent and well-typed JSON output**.

**5. Better Interoperability with Other Tools**

* Tools like **Hive, Trino, and Superset** prefer structured, schema-based data.
* Parquet + Avro schema ensures **type consistency** and **column clarity** for these tools.

**🔍 Summary**

| **Without AvroSchemaRegistry** | **With AvroSchemaRegistry** |
| --- | --- |
| Quick and simple conversion | Structured, validated conversion |
| Schema inferred automatically | Schema explicitly defined and reusable |
| Less control over data types | Full control over data types |
| Harder to handle schema changes | Easy to manage schema evolution |

**✅ Conclusion:**

* You **can** skip the AvroSchemaRegistry if your task is simple.
* But in **production-ready pipelines**, it's **best practice** to use it for **consistency, validation, and reuse**, especially when working with partitioned data and downstream tools like Hive.

Would you like a visual or simplified example showing the difference with and without schema registry?

## Q (ChatGPT). AVRO & Parquet ?

=> 2nd question is why the there are some parquet-related steps referenced and mentioned in the solution in the step below and why they are needed - I am confused as "Avro" was mentioned in the previous step for the "Store the Data" step shown below,

Store Data:

• Use another UpdateAttribute processor to set dynamic output paths and filenames for the partitioned data.

o Create a custom attribute filename with a value using NiFi Expression Language: ${uuid()}.snappy.parquet. This ensures unique filenames for each output file, indicating Parquet format with Snappy compression.

o Create a custom attribute output.path with the value: type=${type}. This dynamically creates HDFS-compatible partition paths (e.g., /user/tutorial/casestudy/netflix/type=Movie/ or /user/tutorial/casestudy/netflix/type=TV Show/).

• Utilize a PutParquet processor to write the partitioned JSON data to HDFS in Parquet format.1

 Configure the PutParquet processor with a JsonTreeReader (set to infer schema) as the record reader.

## Answer

Great question — you're picking up on an important distinction between **Avro** and **Parquet**, and how both show up in the pipeline at different stages. Let’s clarify why **Avro is mentioned in the preparation step**, but **Parquet is used in the storage step**, and why this is **not a contradiction**, but actually **a practical design choice**.

**✅ Quick Summary of the Confusion**

* Step 2 ("Prepare Data") uses **AvroSchemaRegistry** — so you’re thinking **Avro format?**
* But in Step 3 ("Store Data"), it stores data as **Parquet** using **PutParquet**, not Avro.

So why the mix? Why involve Avro if we’re not saving the data as Avro files?

**🧠 Explanation**

**🔷 What is AvroSchemaRegistry really doing?**

* It’s **not writing Avro files**.
* It’s **only providing a schema definition** for use by NiFi processors.
* Think of it as a **blueprint** that says:

“Here’s the structure of your data — the expected fields, types, and nullability.”

* The **JSON writer** (JsonRecordSetWriter) uses this schema to produce **well-structured JSON** data before storing.

So even though it’s an *Avro* registry, it’s **used for schema reference**, not for file format.

**🔶 Why store the data as Parquet?**

Parquet is a **columnar storage format**, which is better for analytics, especially when querying subsets of columns.

**Benefits of storing as Parquet:**

* **Efficient storage** (smaller files with compression, like Snappy).
* **Faster queries** in tools like Hive and Trino.
* **Column pruning** — only read the needed columns.
* **Standard format** for analytics platforms (Hive, Spark, Presto, Superset).

**🔄 How it all fits together:**

| **Step** | **Tool/Processor** | **Format/Function** |
| --- | --- | --- |
| Prepare | CSVReader + AvroSchemaRegistry + JsonRecordSetWriter | Read CSV ➜ Validate against Avro schema ➜ Output as JSON |
| Store | JsonTreeReader + PutParquet | Read structured JSON ➜ Write as **Parquet** to HDFS |

So the flow is:  
CSV ➜ JSON (with schema enforcement) ➜ Parquet

**✅ Why not just use Avro format directly?**

You could. But **Parquet** is preferred in analytics contexts because:

* It’s more **efficient** for read-heavy workloads.
* It plays better with **Hive, Trino, Superset**, etc.
* It's a **de facto standard** in modern data lakes.

Avro is more useful when:

* You’re streaming data.
* You need schema evolution features across systems.
* You’re integrating with Kafka or doing row-based processing.

**🎯 Final Answer:**

* **AvroSchemaRegistry** is used *only to define and enforce the schema*.
* **Parquet** is chosen as the final **storage format** because it’s better suited for analytics.
* NiFi **reads structured JSON** (thanks to Avro schema), then **writes to Parquet** using a JsonTreeReader + PutParquet.

Let me know if you'd like a diagram showing this end-to-end.

## Q (ChatGPT) – Avro & Parquet (2)

but why Parquet was not used in both stages/steps of the solution : for define the schema first and then to be sued as the final storage format

## Answer

That's a **very insightful question**, and it's one that highlights the **division of responsibility between schema handling and storage format** in tools like NiFi.

Let’s break it down clearly:

**✅ Short Answer:**

**Parquet is a file format** used for storing data efficiently.  
**AvroSchemaRegistry in NiFi is for managing schemas, not for file formats.**

You can't use **Parquet** to define a schema in NiFi's schema services — instead, **NiFi uses Avro format to manage schemas** even when you're writing the data later as **Parquet** files.

**🔍 Let’s Explain It Step by Step**

**🔷 Why Parquet is not used to define the schema:**

* **Parquet** is a **binary, columnar file format** — it stores both data and schema together, but:
  + In NiFi, **there's no "ParquetSchemaRegistry"**.
  + NiFi’s schema-aware components (like PartitionRecord, RecordReader, RecordWriter) rely on **Avro** as the common schema definition language.
  + Even if the final file is Parquet, **the schema used during transformation is defined using Avro format.**

Think of it like this:  
**Avro = Schema language** (used to validate and structure data inside NiFi)  
**Parquet = Storage format** (used to store the data at the end, efficiently)

**🔶 Why Parquet is used only at the storage step:**

* Once the data is validated and structured using Avro schema, you can choose **any output format** NiFi supports:
  + JSON
  + CSV
  + Avro
  + **Parquet**

Parquet is picked at the end because:

* It’s highly compressed.
* It’s columnar (great for analytics).
* Tools like Hive, Trino, and Superset **expect** or perform best with Parquet.

**🧠 Real-World Analogy**

Think of it like this:

| **Role** | **Tool** | **Analogy** |
| --- | --- | --- |
| Schema Definition | AvroSchemaRegistry | A blueprint used to guide the construction of a building |
| Intermediate Format | JSON (with schema) | The temporary form your data takes during transformation |
| Final Storage Format | Parquet | The durable building you end up storing and using |

Just like an architect uses a **blueprint (schema)** to guide construction, NiFi uses **Avro schemas** to ensure the structure of data, no matter how it is eventually saved.

**✅ Summary**

| **Feature** | **AvroSchemaRegistry** | **Parquet File** |
| --- | --- | --- |
| Purpose | Define data schema (structure, types) | Store data efficiently (binary, columnar) |
| Used for | Validation, transformation in NiFi | Final storage on HDFS |
| Format | Text-based Avro schema (.avsc) | Binary file format (.parquet) |
| Used in which step | "Prepare Data" step | "Store Data" step |

**🔄 Could you store data in Avro instead of Parquet?**

Yes, you can! NiFi supports PutAvro processor too. But:

* **Parquet is typically better for analytics** tools.
* Parquet + Snappy compression is a **modern default** in data lake pipelines.

Let me know if you’d like a simplified flowchart showing this pipeline!

## Q (ChatGPT) – JSON

so is not it possible and valid to go directly from "Define data schema (structure, types)" using the "AvroSchemaRegistry" to the step for "Store data efficiently (binary, columnar)" using "Parquet File" without passing through the Intermediate Format / JSON (with schema)

## Answer

Excellent follow-up — you're asking **exactly the right question** that draws the boundary between **what’s technically possible** and what’s **currently supported by NiFi**.

**✅ Short Answer:**

**Yes**, conceptually it **should be possible** to go directly from **AvroSchemaRegistry ➜ Parquet output**, **skipping JSON as an intermediate format**,  
**but in NiFi**, this is only possible if the **Parquet writer (PutParquet)** is paired with a **RecordReader** that supports schema registry (like AvroReader) — **and that’s where the limitation usually appears**.

**🔍 Let’s Explain It Clearly**

**✅ What You Want to Do (Ideal Flow):**

CSV ➜ Apply Avro Schema (via AvroSchemaRegistry) ➜ Directly write to Parquet

This would mean:

* Using a **CSVReader** (to read CSV),
* An **AvroSchemaRegistry** (to apply the schema), and
* A **Parquet writer** that can directly use that schema (without needing to go through JSON).

**🔄 What NiFi Actually Does:**

NiFi requires:

* A **RecordReader** to read structured records.
* A **RecordWriter** (or writer-supporting processor) to write those records.

In your case:

* You use a **JsonRecordSetWriter** (which uses Avro schema for structure) → intermediate JSON.
* Then, use a **JsonTreeReader** in the PutParquet processor → reads that JSON and writes to Parquet.

This two-step approach is needed **because NiFi doesn’t currently offer a direct Avro-to-Parquet bridge** inside a single processor using schema registry.

**🔧 Could You Do It in One Step in NiFi?**

**✅ Technically Yes, *If*:**

You configure the PutParquet processor with:

* A **CSVReader** (to read the raw CSV input),
* And an **AvroSchemaRegistry** (used by the CSVReader for schema),
* So the processor can go:
* CSV ➜ Schema (Avro) ➜ Parquet (without JSON in between)

But in practice:

* The **PutParquet processor** expects to receive data already in **record format** from upstream (often JSON),
* And **JsonTreeReader** is the only RecordReader that easily supports schema inference and complex nested types that Parquet expects.

So the typical NiFi pattern is:

CSV ➜ JSON with schema ➜ Parquet

rather than:

CSV ➜ Parquet directly using AvroSchemaRegistry

**✅ Conclusion:**

| **Flow** | **Possible in NiFi?** | **Recommended/Typical?** | **Notes** |
| --- | --- | --- | --- |
| CSV ➜ AvroSchemaRegistry ➜ JSON ➜ Parquet | ✅ Yes | ✅ Yes | This is what you’ve implemented. |
| CSV ➜ AvroSchemaRegistry ➜ Parquet (no JSON) | 🔶 Possible in theory | 🔶 Uncommon in practice | Needs custom wiring and may be limited by NiFi’s RecordReader compatibility. |
| CSV ➜ Parquet (no schema registry) | ✅ Yes (with inference) | ❌ Risky for production | You lose schema validation and type control. |

Would you like a visual diagram showing these paths side-by-side for clarity?

## Q. (ChatGPT) “Controller service” VS. “Processor “ in NiFi

What is the difference between "controller service" and "Processor" in NiFi or are both of them the same or is one of them a subtype of the other ?

## Answer

Great question! **"Controller Services"** and **"Processors"** in Apache NiFi are **not the same** — they serve **complementary but distinct purposes** in the data flow architecture. Let’s break it down simply and clearly:

**🧱 1. What is a NiFi Processor?**

**🔧 Definition:**

A **Processor** is an **active building block** in NiFi that performs a specific task in the data pipeline — like reading a file, transforming content, routing data, or writing to a system.

**📦 Examples of Processors:**

* GetFile – reads a file from the local filesystem.
* PutParquet – writes data to HDFS in Parquet format.
* PartitionRecord – splits records based on field values.
* UpdateAttribute – modifies FlowFile metadata (attributes).

**⚙️ Key Characteristics:**

* Processors are placed **on the canvas** and **actively handle data**.
* They process **FlowFiles**, perform logic, and pass them downstream.
* Each processor has a **configure → run → transfer** lifecycle.

**🧩 2. What is a Controller Service?**

**🔧 Definition:**

A **Controller Service** is a **reusable, shared configuration component** that provides services like connecting to external systems or defining data formats — but it **doesn’t move or transform data by itself**.

**📦 Examples of Controller Services:**

* CSVReader – tells processors how to parse incoming CSV data into records.
* AvroSchemaRegistry – stores and manages reusable Avro schemas.
* DBCPConnectionPool – defines a database connection that processors can use.
* JsonRecordSetWriter – writes structured records in JSON format.

**⚙️ Key Characteristics:**

* They **don’t appear directly in the flow** — you configure them **inside processors**.
* They are **shared across multiple processors** to ensure consistency.
* Typically, they support **Record-oriented processors** (like PutParquet, PartitionRecord, etc.)

**🧠 Key Differences Summary**

| **Feature** | **Processor** | **Controller Service** |
| --- | --- | --- |
| **Role** | Does the actual data work (read, write, transform) | Provides configuration or supporting services |
| **Visible on Canvas** | ✅ Yes | ❌ No |
| **Moves/Processes Data** | ✅ Yes | ❌ No |
| **Reusable Across Processors** | Sometimes (via templates) | ✅ Yes |
| **Examples** | GetFile, UpdateAttribute, PutParquet | CSVReader, AvroSchemaRegistry, JsonRecordSetWriter |
| **Depends on Other** | ✅ Often uses Controller Services | ❌ Does not depend on Processors |

**🧠 Analogy:**

Imagine building a factory pipeline:

* **Processors** are like the **machines** doing the work — they cut, paint, assemble.
* **Controller Services** are like the **tools, blueprints, or shared control panels** that multiple machines use (e.g., a shared paint color config or a standard screw size).

**✅ Conclusion:**

* They are **not the same**, and **neither is a subtype** of the other.
* **Processors = workers doing the task**
* **Controller Services = tools/configurations that help the workers**

Both are essential, and they **work together** to build powerful, reusable, and efficient NiFi data flows.

Would you like a visual example showing a processor and its controller services in a sample flow?

## Q (ChatGPT) – Tale/Narrative Fashion

Great - now provide the steps below in a narrative-like storytelling fashion. You can start with something like "Once upon a time, in the land of NiFi, The data engineering team found themsleves self almost lost in face of a great chanllenge. They were asked to .... and they decided to plan for their way of work by asking very basic primitive questions, one at a time, for each question, they would first prepare a simple answer followed by the tools (NiFi processor , NiFi controller Service ,...) to use documenting why and they would use it, First to "Ingest the Data (from Local File System)", a ... would be required, ..."

🔁 1. Ingest the Data (from Local File System)

• Use the GetFile processor to read the "Netflix Titles.csv" from a local folder.

• Configure it to pick only that file, and to delete/move it after processing (to avoid reprocessing).

• Since this is a static file, it's typically a one-time read, but you can set a schedule (e.g., every 10 minutes) for automation or re-runs.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

🛠️ 2. Prepare the Data (Parse, Structure, and Partition)

• Use UpdateAttribute to add a custom attribute:

o schema.name = NetflixTitleSchema (used later in schema registry).

• Use a PartitionRecord processor to split the data by content type (Movie vs TV Show).

• Set up a CSVReader (controller service):

o Enable “Treat First Line as Header” to correctly map CSV columns.

• Set up an AvroSchemaRegistry (controller service):

o Add a schema called NetflixTitleSchema with all expected fields (e.g., show\_id, title, director).

o Allow nulls for optional fields (like cast or director).

• Set up a JsonRecordSetWriter (controller service):

o Use it to write parsed data in JSON format, referencing the NetflixTitleSchema.

• In PartitionRecord, configure it to partition based on the type field

(i.e., “Movie” or “TV Show”), creating separate flow files for each.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

💾 3. Store the Data (Save to HDFS in Parquet Format)

• Use UpdateAttribute to define dynamic file names and output paths:

o Set filename = ${uuid()}.snappy.parquet (generates a unique Parquet file name).

o Set output.path = type=${type} (creates dynamic folders like /type=Movie/).

• Use the PutParquet processor to write data to HDFS in Parquet format:

o Use a JsonTreeReader with schema inference enabled to read the records.

o Use the ${output.path} in the output directory path

(e.g., /user/tutorial/casestudy/netflix/${output.path}).

o Set compression to Snappy for efficient storage.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

🧠 4. Enable Querying and Build Dashboard (Hive, Trino, Superset)

• In Hive, create:

o A new database and

o An external table for Netflix data, partitioned by the type column.

o Point the table location to the HDFS path used in NiFi.

• Run MSCK REPAIR TABLE to load and refresh partitions in Hive.

• Use Trino to query the Hive table for analysis.

• Use a BI tool like Apache Superset to create dashboards and visualizations from the data.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

✅ Key Benefits of This NiFi Workflow

• Automated ingestion and structured transformation of raw CSV data.

• Efficient partitioning for better performance.

• Optimized storage using Parquet + Snappy on HDFS.

• Seamless integration with Hive, Trino, and dashboarding tools.Answer

Sure! Here's your NiFi workflow transformed into a **narrative-style story**, full of personality and clarity — perfect for storytelling in a classroom or presentation setting.

**📖 The Tale of the Netflix Data Journey in the Land of NiFi**

**Once upon a time**, in the land of **NiFi**, a brave data engineering team faced a challenge that echoed through the pipelines:  
They were asked to **ingest, transform, store, and visualize** a mysterious scroll of information — a file called **"Netflix Titles.csv"**.

The team knew that to succeed, they had to move carefully — asking one **simple question at a time**, planning their journey like explorers mapping unknown lands. For each question, they would decide on:

* ✅ The task to complete
* 🔧 The tools they would use (NiFi Processors, Controller Services)
* 🧠 The reason why that tool was chosen

**🔁 Chapter 1: Ingest the Data (from the Local File System)**

**The First Question:**  
*"How do we get this ancient scroll (CSV file) into our NiFi flow?"*

**The Answer:**  
They needed a tool that could **pick up a file from a specific local folder**, just like a messenger retrieving a scroll from a stone chest.

**The Tool:**  
🛠️ **Processor:** GetFile

* It would **watch a folder**, grab the Netflix Titles.csv, and move it out of the way after reading it (to avoid rereading it again and again).
* Even though this scroll wasn’t changing, they added a schedule — every 10 minutes — just in case it were ever replaced.

**🛠️ Chapter 2: Prepare the Data (Parse, Structure, and Partition)**

**The Second Question:**  
*"Now that we have the scroll, how do we understand its contents and separate its stories?"*

**The Answer:**  
They would **decode the CSV**, map the fields to a known schema, and **split it into two flows** — one for **Movies**, one for **TV Shows**.

**The Tools:**

1. 🛠️ **Processor:** UpdateAttribute
   * To mark the scroll with a tag: schema.name = NetflixTitleSchema
   * This would later help readers know how to interpret the text.
2. 🧩 **Controller Service:** CSVReader
   * The team activated a service that knew how to read CSV scrolls.
   * It was told, “Treat the first line as a header,” because that line had the names of the fields.
3. 🧠 **Controller Service:** AvroSchemaRegistry
   * They carefully wrote a schema: the list of expected fields like show\_id, title, director, cast, release\_year, etc.
   * Optional values (like cast and director) were allowed to be blank.
   * This schema ensured all scrolls would be read the same way, no matter who read them.
4. 🧩 **Controller Service:** JsonRecordSetWriter
   * Once decoded, the scroll would be rewritten in a new language: **JSON**.
   * This format would be easier to store and more compatible with modern analytics tools.
5. 🛠️ **Processor:** PartitionRecord
   * This wise processor looked at the type field in each row — and **divided the records into two flows**:
     + One for **TV Shows**
     + One for **Movies**

And with that, the scroll was not only readable but organized for the journey ahead.

**💾 Chapter 3: Store the Data (Save to HDFS in Parquet Format)**

**The Third Question:**  
*"Where do we store our precious, structured knowledge?"*

**The Answer:**  
The team chose to send their records to the **great vault of HDFS**, storing them as **Parquet files**, compressed with **Snappy magic** for efficiency.

**The Tools:**

1. 🛠️ **Processor:** UpdateAttribute
   * They gave each output file a **unique name**: ${uuid()}.snappy.parquet
   * And they crafted a **dynamic folder path** using the record type: type=${type}  
     (e.g., /type=Movie/ or /type=TV Show/)
2. 🛠️ **Processor:** PutParquet
   * This powerful writer took the JSON records and saved them as **Parquet files**.
   * To read the JSON, it used a **JsonTreeReader** with schema inference turned on.
   * It stored the files under: /user/tutorial/casestudy/netflix/${output.path}
   * And it ensured Snappy compression for speed and space-saving.

The scrolls were now transformed, organized, and safely tucked away in the vault — ready for questioning.

**🧠 Chapter 4: Query the Data and Reveal Insights**

**The Fourth Question:**  
*"How do we let the kingdom see the stories in this data?"*

**The Answer:**  
They built a pathway to querying and visualization, using the tools of the analysts and dashboard builders.

**The Tools:**

1. 🐝 **Hive**
   * A database was created, and an **external table** was defined pointing to the HDFS path.
   * The table was **partitioned by the type column** so that movies and TV shows were easy to query separately.
   * They ran the ancient command: MSCK REPAIR TABLE to tell Hive about all the partition folders.
2. 🔍 **Trino**
   * This powerful query engine connected to Hive and allowed fast, SQL-like exploration of the data.
3. 📊 **Apache Superset**
   * With data flowing in, the team built **beautiful dashboards** to show trends, counts, and insights — making it easy for everyone to understand the Netflix universe.

**✅ Epilogue: The Triumph of the NiFi Team**

The team had succeeded. By:

* Automating the ingestion of structured data
* Partitioning it cleanly for optimized querying
* Storing it efficiently in a modern format
* And visualizing it with clarity

They turned raw data into **valuable knowledge** — and proved that with **NiFi’s processors and controller services**, even the most ancient scrolls could tell modern stories.

Let me know if you’d like to visualize this journey as a diagram or animation — we can bring the tale to life!

## Q. (ChatGPT) – HDFS ?

why storing data into HDFS was required ? was it for partitioning purpose only ? or are there any different other purposes that cannot be achieved by directly handling and working with the files as they are stored as normal/regular on linux or other OS ?

## Answer

Fantastic question — and it touches on a **key architectural choice** in big data systems: **why store data in HDFS (Hadoop Distributed File System)** instead of just working with **regular files on Linux or local OS storage**.

Let’s break this down step-by-step:

**❓ Is HDFS used *only* for partitioning?**

**No.** Partitioning is **one benefit**, but not the **main reason** for using HDFS.

**🧠 So why is HDFS used? What does it offer that local files can’t?**

Here are the **main reasons** for using HDFS over regular local file storage:

**1. 💽 Scalability Across Many Machines**

* **HDFS spreads data across many servers**, not just one.
* A single CSV or Parquet file might be split into blocks and stored across a **distributed cluster**.
* This makes it possible to store and process **huge datasets** (terabytes or petabytes) — far beyond what a single Linux machine could handle.

**2. ⚡ Fault Tolerance & Data Replication**

* In HDFS, each data block is **replicated** (e.g., 3 copies by default) on different machines.
* If a disk or server fails, data is **not lost** — it can be recovered automatically.
* Local file systems **do not have built-in replication or recovery** features.

**3. 🔄 Optimized for Distributed Processing**

* HDFS is tightly integrated with tools like **Hive, Trino, Spark, and MapReduce**.
* These tools can read from HDFS in **parallel**, speeding up processing.
* With local files, parallel access is much harder to coordinate.

**4. 📂 Supports Partitioned Data Layouts Natively**

* Yes, **partitioning is easier** in HDFS because it's part of the design.
* Tools like Hive automatically recognize folder structures like type=Movie and use them to **optimize query performance**.
* While you *could* create folders on a local disk, it **wouldn’t integrate** as cleanly with Hive or Trino.

**5. 🛠️ Centralized Storage for Shared Access**

* In a team or cluster setup, **HDFS acts as a shared data lake**.
* Multiple users and tools can access the same data without needing to copy files across machines.
* With local storage, each user or process might need its own copy of the data.

**6. 📈 Built for High Throughput (Not Low Latency)**

* HDFS is optimized for **large file reads/writes**, not quick small reads.
* This makes it ideal for **batch processing** of analytics data (like Parquet files).
* Local filesystems are better at interactive, low-latency reads (but that’s not the use case here).

**🔍 Summary: HDFS vs. Local Files**

| **Feature** | **HDFS** | **Local Filesystem** |
| --- | --- | --- |
| Scales across machines | ✅ Yes | ❌ No |
| Fault tolerance | ✅ Yes (replication) | ❌ No |
| Works with Hive/Trino/Spark | ✅ Native integration | ⚠️ Requires extra setup |
| Partitioned data folders | ✅ Optimized and expected | ⚠️ Manual and limited use |
| Shared access for teams | ✅ Yes (centralized) | ❌ Not designed for multi-user |
| High throughput for big files | ✅ Yes | ⚠️ Limited to machine resources |
| Requires Hadoop setup | ⚠️ Yes | ✅ No setup needed |

**✅ So, when *should* you use HDFS?**

Use HDFS when:

* You're dealing with **large datasets** (big data).
* You want to **query partitioned data** efficiently with Hive, Trino, or Spark.
* You need **reliable, scalable, shared storage** for a team or cluster.
* You plan to run **analytics pipelines** or dashboards at scale.

Use local files only when:

* You're doing **lightweight processing**, testing, or prototyping.
* You don’t need scalability or integration with Hive/Spark.
* You're working **alone** on a small machine.

Let me know if you’d like a visual chart comparing HDFS and local storage or a simple story-based analogy!