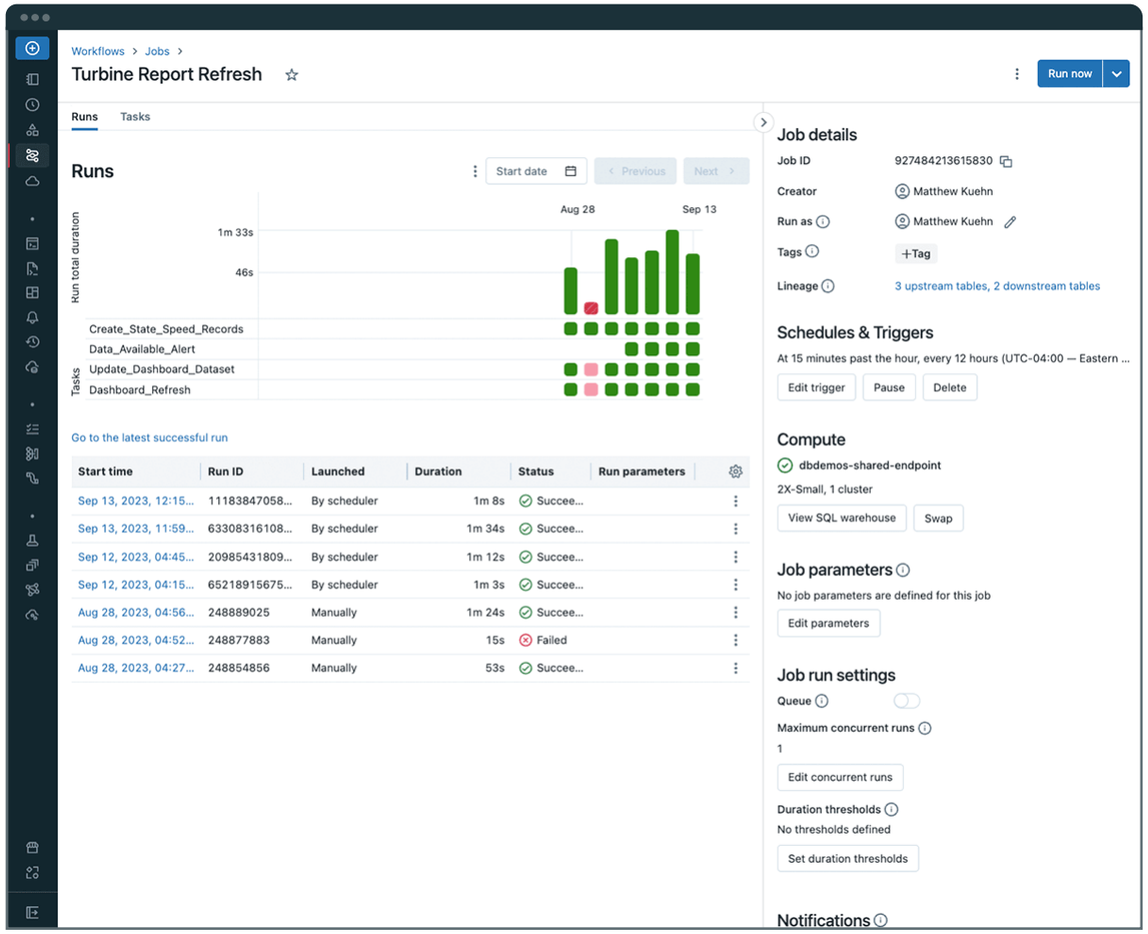
MONITORING YOUR PRODUCTION PIPELINES

While authoring is comprehensive within Databricks Workflows, it is only one part of the picture. Equally important is the ability to easily monitor and debug your pipelines once they are built and in production.



Databricks Workflows allows users to monitor individual job runs, offering insights into task outcomes and overall execution times. This visibility helps analysts understand query performance, identify bottlenecks and address issues efficiently. By promptly recognizing tasks that require attention, analysts can ensure seamless data processing and quicker issue resolution.

When it comes to executing a pipeline at the right time, Databricks Workflows allows users to schedule jobs for execution at specific intervals or trigger them when certain files arrive. In the above image, we were first manually triggering this pipeline to test and debug our tasks. Once we got this to a steady state, we began triggering this every 12 hours to accommodate for data refresh needs across time zones. This flexibility accommodates varying data scenarios, ensuring timely pipeline execution. Whether it's routine processing or responding to new data batches, analysts can tailor job execution to match operational requirements.

Late-arriving data can bring a flurry of questions to a data analyst from end users. Workflows enables analysts and consumers alike to stay informed on data freshness by setting up notifications for job outcomes such as successful execution, failure or even a long-running job. These notifications ensure timely awareness of changes in data processing. By proactively evaluating a pipeline's status, analysts can take proactive measures based on real-time information.

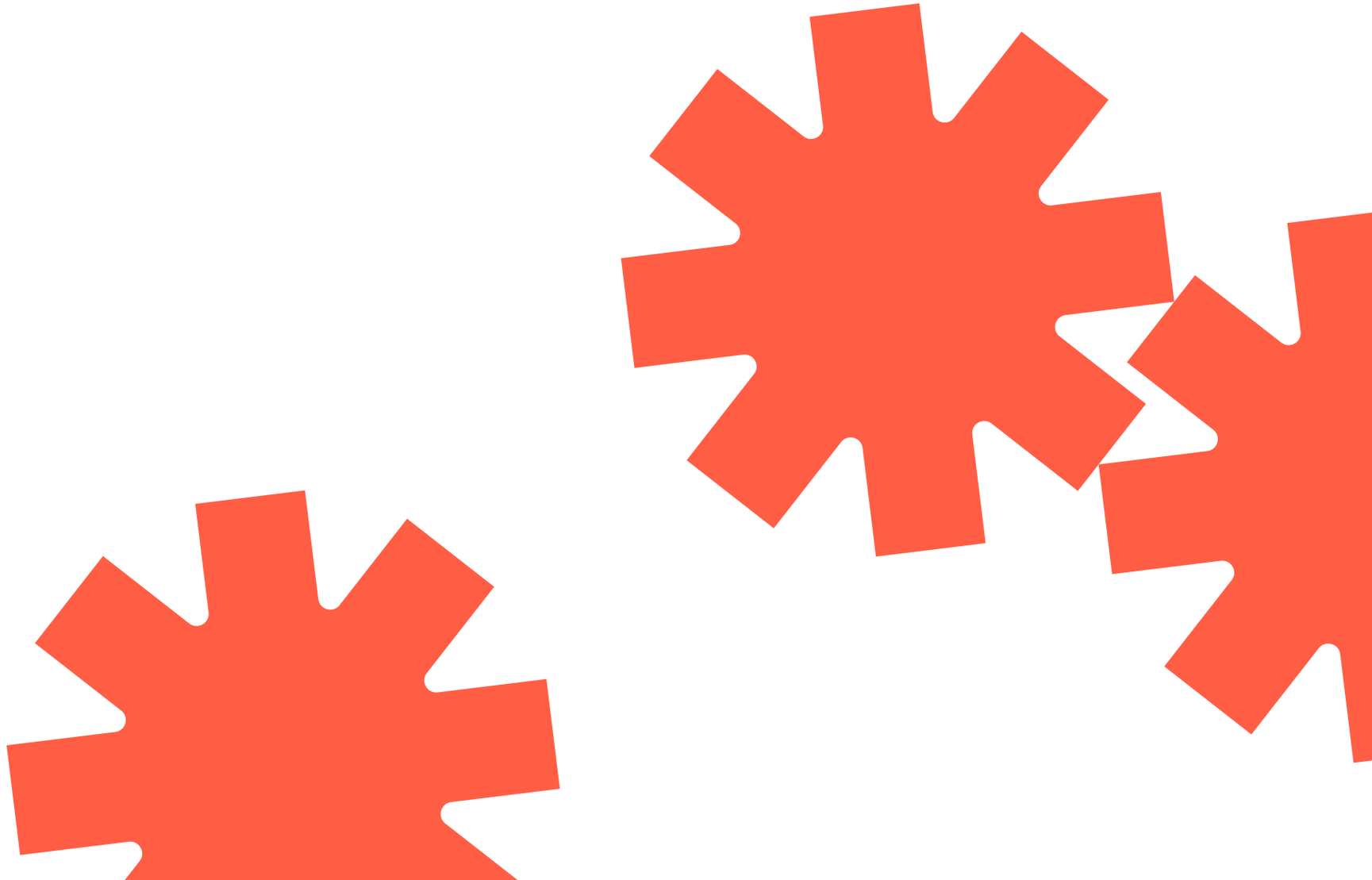
As with all pipelines, failures will inevitably happen. Workflows helps manage this by allowing analysts to configure job tasks for automatic retries. By automating retries, analysts can focus on generating insights rather than troubleshooting intermittent technical issues.

CONCLUSION

In the evolving landscape of data analysis tools, Databricks Workflows bridges the gap between data analysts and the complexities of data orchestration. By automating tasks, ensuring data quality and providing a user-friendly interface, Databricks Workflows empowers analysts to focus on what they excel at — extracting meaningful insights from data. As the concept of the lakehouse continues to unfold, Workflows stands as a pivotal component, promising a unified and efficient data ecosystem for all personas.

GET STARTED

- Learn more about Databricks Workflows
- Take a product tour of Databricks Workflows
- Create your first workflow with this quickstart guide



Schema Management and Drift Scenarios via Databricks Auto Loader

by Garrett Peternel

Data lakes notoriously have had challenges with managing incremental data processing at scale without integrating open table storage format frameworks (e.g., Delta Lake, Apache Iceberg, Apache Hudi). In addition, schema management is difficult with schema-less data and schema-on-read methods. With the power of the Databricks Platform, Delta Lake and Apache Spark provide the essential technologies integrated with Databricks Auto Loader (AL) to consistently and reliably stream and process raw data formats incrementally while maintaining stellar performance and data governance.

AUTO LOADER FEATURES

AL is a boost over Spark Structured Streaming, supporting several additional benefits and solutions including:

- Databricks Runtime only Structured Streaming cloudFiles source
- Schema drift, dynamic inference and evolution support
- Ingests data via JSON, CSV, PARQUET, AVRO, ORC, TEXT and BINARYFILE input file formats
- Integration with cloud file notification services (e.g., Amazon SQS/SNS)
- Optimizes directory list mode scanning performance to discover new files in cloud storage (e.g., AWS, Azure, GCP, DBFS)

For further information please visit the official Databricks Auto Loader documentation.

SCHEMA CHANGE SCENARIOS

In this chapter I will showcase a few examples of how AL handles schema management and drift scenarios using a public IoT sample dataset with schema modifications to showcase solutions. Schema 1 will contain an IoT sample dataset schema with all expected columns and expected data types. Schema 2 will contain unexpected changes to the IoT sample dataset schema with new columns and changed data types. The following variables and paths will be used for this demonstration along with Databricks Widgets to set your username folder.

```scala
%scala
dbutils.widgets.text("dbfs_user_dir", "your_user_name") // widget for account email

val userName = dbutils.widgets.get("dbfs_user_dir")
val rawBasePath = s"dbfs:/user/$userName/raw/"
val repoBasePath = s"dbfs:/user/$userName/repo/"

val jsonSchema1Path = rawBasePath + "iot-schema-1.json"
val jsonSchema2Path = rawBasePath + "iot-schema-2.json"
val repoSchemaPath = repoBasePath + "iot-ddl.json"

dbutils.fs.rm(repoSchemaPath, true) // remove schema repo for demos
```

SCHEMA 1

```scala
%scala
spark.read.json(jsonSchema1Path).printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
```

```
%scala
display(spark.read.json(jsonSchema1Path).limit(10))
```

Table + New result table: OFF

	alarm_status	battery_level	c02_level	cca2	cca3	cn	coordinates	date	device_id	device_serial_num
1	yellow	3	1082	US	USA	United States	{"latitude": 38, "longitude": -97}	2016-03-20	62	62fH8oKr8aiT
2	green	0	920	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	241	241un29KmR
3	yellow	7	1004	US	USA	United States	{"latitude": 42.36, "longitude": -71.05}	2016-03-20	260	260F7DTEbnz
4	yellow	9	1081	DE	DEU	Germany	{"latitude": 51.45, "longitude": 7.02}	2016-03-20	298	298hJceoQv0
5	yellow	8	1311	US	USA	United States	{"latitude": 38.88, "longitude": -92.4}	2016-03-20	301	301wHcyNzw
6	red	1	1484	IN	IND	India	{"latitude": 20, "longitude": 77}	2016-03-20	334	334DUAg4mbXi
7	yellow	9	1266	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	349	349gRZSGtcGR
8	yellow	1	1085	GB	GBR	United Kingdom	{"latitude": 51.51, "longitude": -0.09}	2016-03-20	383	383yNMmfRq0zG
9	red	3	1508	GB	GBR	United Kingdom	{"latitude": 53.5, "longitude": -2.19}	2016-03-20	391	391Qn3lLA
10	yellow	5	1191	KR	KOR	Republic of Korea	{"latitude": 37.29, "longitude": 127.01}	2016-03-20	455	455L4J9FUG

SCHEMA 2

```
%scala
// NEW => device_serial_number_device_type, location
spark.read.json(jsonSchema2Path).printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number_device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: double (nullable = true)
|-- ip: string (nullable = true)
|-- latitude: double (nullable = true)
|-- location: struct (nullable = true)
| |-- cca2: string (nullable = true)
| |-- cca3: string (nullable = true)
| |-- cn: string (nullable = true)
|-- longitude: double (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
```

```
%scala
display(spark.read.json(jsonSchema1Path).limit(10))
```

Table + New result table: OFF

	device_serial_number_device_type	epoch_time_miliseconds	humidity	ip	latitude	location	longitude	scale
1	{"2n2Pea":"sensor-pad"}	1458444054119	70	213.161.254.1	62.47	{"cca2": "NO", "cca3": "NOR", "cn": "Norway"}	6.15	Fahre
2	{"12Y2klm0o":"sensor-pad"}	1458444054126	92	68.28.91.22	38	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-97	Fahre
3	{"230lupA":"meter-gauge"}	1458444054133	47	59.90.65.1	12.98	{"cca2": "IN", "cca3": "IND", "cn": "India"}	77.58	Fahre
4	{"36VQv8fnEg":"sensor-pad"}	1458444054141	47	213.7.14.1	35	{"cca2": "CY", "cca3": "CYP", "cn": "Cyprus"}	33	Fahre
5	{"448DeWGL":"sensor-pad"}	1458444054149	63	62.128.16.74	49.46	{"cca2": "DE", "cca3": "DEU", "cn": "Germany"}	11.1	Fahre
6	{"49YesGXwt":"meter-gauge"}	1458444054152	70	170.37.224.1	42.28	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-71.44	Fahre
7	{"64djcln":"sensor-pad"}	1458444054162	55	38.99.198.186	38	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-97	Fahre
8	{"77IKW3YAB55":"meter-gauge"}	1458444054169	82	218.248.255.30	12.98	{"cca2": "IN", "cca3": "IND", "cn": "India"}	77.58	Fahre
9	{"80TY4dWSMH":"sensor-pad"}	1458444054171	57	159.128.0.181	50.01	{"cca2": "CA", "cca3": "CAN", "cn": "Canada"}	-97.22	Fahre
10	{"94HL9jChD":"sensor-pad"}	1458444054179	66	24.32.26.1	39.33	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-120.25	Fahre

EXAMPLE 1: SCHEMA TRACKING/MANAGEMENT

AL tracks schema versions, metadata and changes to input data over time via specifying a location directory path. These features are incredibly useful for tracking history of data lineage, and are tightly integrated with the Delta Lake transactional log DESCRIBE HISTORY and Time Travel.

```
%scala
val rawAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath) // schema history tracking
.load(jsonSchema1Path)
)
```

```
%scala
rawAlDf.printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: string (nullable = true)
|-- c02_level: string (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: string (nullable = true)
|-- date: string (nullable = true)
|-- device_id: string (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: string (nullable = true)
|-- humidity: string (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: string (nullable = true)
|-- timestamp: string (nullable = true)
|-- _rescued_data: string (nullable = true)
```

```
%scala
display(rawAlDf.limit(10))
```

By default (for JSON, CSV and XML file format) AL infers all column data types as strings, including nested fields.

Table + New result table: OFF

	alarm_status	battery_level	c02_level	cca2	cca3	cn	coordinates	date	device_id	device_serial_
1	yellow	3	1082	US	USA	United States	{"latitude":38.0,"longitude":-97.0}	2016-03-20	62	62fH8oKr8aiT
2	green	0	920	US	USA	null	{"latitude":47.0,"longitude":8.0}	2016-03-20	241	241un29KmR
3	yellow	7	1004	US	USA	United States	{"latitude":42.36,"longitude":-71.05}	2016-03-20	260	260F7DTEbnz
4	yellow	9	1081	DE	DEU	Germany	{"latitude":51.45,"longitude":7.02}	2016-03-20	298	298hJceoQv0
5	yellow	8	1311	US	USA	United States	{"latitude":38.88,"longitude":-92.4}	2016-03-20	301	301wHcyNzw
6	red	1	1484	IN	IND	India	{"latitude":20.0,"longitude":77.0}	2016-03-20	334	334DUAg4mb
7	yellow	9	1266	US	USA	null	{"latitude":47.0,"longitude":8.0}	2016-03-20	349	349gRZSGtcG
8	yellow	1	1085	GB	GBR	United Kingdom	{"latitude":51.51,"longitude":-0.09}	2016-03-20	383	383yNMmfRq0
9	red	3	1508	GB	GBR	United Kingdom	{"latitude":53.5,"longitude":-2.19}	2016-03-20	391	391Qn3lLA
10	yellow	5	1191	KR	KOR	Republic of Korea	{"latitude":37.29,"longitude":127.01}	2016-03-20	455	455L4J9FUG

Here is the directory structure where AL stores schema versions. These files can be read via Spark DataFrame API.

Schema Repository

```
%scala
display(dbutils.fs.ls(repoSchemaPath + "/_schemas"))
```

Table + New result table: OFF

	path	name	size	modificationTime
1	dbfs:/user/garrett.peternel@databricks.com/repo/iot-ddl.json/_schemas/0	0	1628	1709306037000

Schema Metadata

```
%scala
display(spark.read.json(repoSchemaPath + "/_schemas"))
```

Table + New result table: OFF

	_corrupt_record	dataSchemaJson	partitionSchemaJson
1	v1	null	null
2	null	{"type":"struct","fields":[{"name":"alarm_status","type":"string","nullable":true,"metadata":{}}, {"name":"battery_level","type":"long","nullable":true,"metadata":{}}, {"name":"c02_level","type":"long","nullable":true,"metadata":{}}, {"name":"cca2","type":"string","nullable":true,"metadata":{}},{"name":"cca3","type":"string","nullable":true,"metadata":{}},{"name":"cn","type":"string","nullable":true,"metadata":{}},{"name":"coordinates","type":{"type":"struct","fields":[{"name":"latitude","type":"do...	{"type":"struct","fields":[]}

EXAMPLE 2: SCHEMA HINTS

AL provides hint logic using SQL DDL syntax to enforce and override dynamic schema inference on known single data types, as well as semi-structured complex data types.

```
%scala
val hintAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.schemaHints", "coordinates STRUCT<latitude:DOUBLE,
longitude:DOUBLE>, humidity LONG, temp DOUBLE") // schema ddl hints
.load(jsonSchema1Path)
)
```

```
%scala
hintAlDf.printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: string (nullable = true)
|-- c02_level: string (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: string (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: string (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
|-- _rescued_data: string (nullable = true)
```

The schema hints specified in the AL options perform the data type mappings on the respective columns. Hints are useful for applying schema enforcement on portions of the schema where data types are known while in tandem with dynamic schema inference covered in Example 3.

```
%scala
display(hintAlDf.limit(10))
```

Table + New result table: OFF

	coordinates	date	device_id	device_serial_number	device_type	epoch_time_miliseconds	humidity	ip	scale
1	{"latitude": 38, "longitude": -97}	2016-03-20	62	62fH8oKr8aiT	sensor-pad	1458444054161	31	151.198.215.1	Fahrenheit
2	{"latitude": 47, "longitude": 8}	2016-03-20	241	241un29KmR	meter-gauge	1458444054259	64	80.84.21.105	Fahrenheit
3	{"latitude": 42.36, "longitude": -71.05}	2016-03-20	260	260F7DTEbnz	sensor-pad	1458444054267	57	72.21.168.109	Fahrenheit
4	{"latitude": 51.45, "longitude": 7.02}	2016-03-20	298	298hJceoQv0	sensor-pad	1458444054280	84	217.76.103.233	Fahrenheit
5	{"latitude": 38.88, "longitude": -92.4}	2016-03-20	301	301wHcyNzw	meter-gauge	1458444054281	26	150.199.7.154	Fahrenheit
6	{"latitude": 20, "longitude": 77}	2016-03-20	334	334DUAg4mbXi	sensor-pad	1458444054292	72	202.71.135.89	Fahrenheit
7	{"latitude": 47, "longitude": 8}	2016-03-20	349	349gRZSGtcGR	meter-gauge	1458444054296	45	195.219.212.9	Fahrenheit
8	{"latitude": 51.51, "longitude": -0.09}	2016-03-20	383	383yNMmfRq0zG	meter-gauge	1458444054308	46	166.49.222.157	Fahrenheit
9	{"latitude": 53.5, "longitude": -2.19}	2016-03-20	391	391Qn3lLA	meter-gauge	1458444054310	97	89.21.16.18	Fahrenheit
10	{"latitude": 37.29, "longitude": 127.01}	2016-03-20	455	455L4J9FUG	therm-stick	1458444054330	64	211.41.134.73	Fahrenheit

EXAMPLE 3: DYNAMIC SCHEMA INFERENCE

AL dynamically searches a sample of the dataset to determine nested structure. This avoids costly and slow full dataset scans to infer schema. The following configurations are available to adjust the amount of sample data used on read to discover initial schema:

1. spark.databricks.cloudFiles.schemaInference.sampleSize.numBytes (default 50 GB)
2. spark.databricks.cloudFiles.schemaInference.sampleSize.numFiles (default 1000 files)

```
%scala
val inferAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.inferColumnTypes", true) // schema inference
.load(jsonSchema1Path)
)
```

```
%scala
inferAlDf.printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
|-- _rescued_data: string (nullable = true)
```

AL saves the initial schema to the schema location path provided. This schema serves as the base version for the stream during incremental processing. Dynamic schema inference is an automated approach to applying schema changes over time.

```
%scala
display(inferAlDf.limit(10))
```

Table + New result table: OFF

	alarm_status	battery_level	c02_level	cca2	cca3	cn	coordinates	date	device_id	device_serial_num
1	yellow	3	1082	US	USA	United States	{"latitude": 38, "longitude": -97}	2016-03-20	62	62fH8oKr8aiT
2	green	0	920	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	241	241un29KmR
3	yellow	7	1004	US	USA	United States	{"latitude": 42.36, "longitude": -71.05}	2016-03-20	260	260F7DTEbnz
4	yellow	9	1081	DE	DEU	Germany	{"latitude": 51.45, "longitude": 7.02}	2016-03-20	298	298hJceoQv0
5	yellow	8	1311	US	USA	United States	{"latitude": 38.88, "longitude": -92.4}	2016-03-20	301	301wHcyNzw
6	red	1	1484	IN	IND	India	{"latitude": 20, "longitude": 77}	2016-03-20	334	334DUAg4mbXi
7	yellow	9	1266	US	USA	null	{"latitude": 47, "longitude": 8}	2016-03-20	349	349gRZSGtcGR
8	yellow	1	1085	GB	GBR	United Kingdom	{"latitude": 51.51, "longitude": -0.09}	2016-03-20	383	383yNMmfRq0zG
9	red	3	1508	GB	GBR	United Kingdom	{"latitude": 53.5, "longitude": -2.19}	2016-03-20	391	391Qn3ILA
10	yellow	5	1191	KR	KOR	Republic of Korea	{"latitude": 37.29, "longitude": 127.01}	2016-03-20	455	455L4J9FUG

EXAMPLE 4: STATIC USER-DEFINED SCHEMA

AL also supports static custom schemas just like Spark Structured Streaming. This eliminates the need for dynamic schema-on-read inference scans, which trigger additional Spark jobs and schema versions. The schema can be retrieved as a DDL string or a JSON payload.

DDL

```
%scala
inferAlDf.schema.toDDL
```

```
String = alarm_status STRING,battery_level BIGINT,c02_level BIGINT,cca2 STRING,cca3
STRING,cn STRING,coordinates STRUCT<latitude: DOUBLE, longitude: DOUBLE>,date
STRING,device_id BIGINT,device_serial_number STRING,device_type STRING,epoch_time_
miliseconds BIGINT,humidity BIGINT,ip STRING,scale STRING,temp DOUBLE,timestamp
STRING,_rescued_data STRING
```

JSON

```
%scala
spark.read.json(repoSchemaPath + "/_schemas").select("dataSchemaJson").
where("dataSchemaJson is not null").first()
```

```
org.apache.spark.sql.Row = [{"type":"struct","fields":[{"name":"alarm_
status","type":"string","nullable":true,"metadata":{}},{"name":"battery_level","type":
"long","nullable":true,"metadata":{}},{"name":"c02_level","type":"long","nullable":true,
"metadata":{}},{"name":"cca2","type":"string","nullable":true,"metadata":{}},{"name":
"cca3","type":"string","nullable":true,"metadata":{}},{"name":"cn","type":"string",
"nullable":true,"metadata":{}},{"name":"coordinates","type":{"type":"struct","fields":
[{"name":"latitude","type":"double","nullable":true,"metadata":{}},{"name":"longitude",
"type":"double","nullable":true,"metadata":{}}]},"nullable":true,"metadata":{}},
{"name":"date","type":"string","nullable":true,"metadata":{}},{"name":"device_id",
"type":"long","nullable":true,"metadata":{}},{"name":"device_serial_number","type":
"string","nullable":true,"metadata":{}},{"name":"device_type","type":"string",
"nullable":true,"metadata":{}},{"name":"epoch_time_miliseconds","type":"long",
"nullable":true,"metadata":{}},{"name":"humidity","type":"long","nullable":true,
"metadata":{}},{"name":"ip","type":"string","nullable":true,"metadata":{}},{"name":
"scale","type":"string","nullable":true,"metadata":{}},{"name":"temp","type":"double",
"nullable":true,"metadata":{}},{"name":"timestamp","type":"string","nullable":true,
"metadata":{}}]}]
```

Here's an example of how to generate a user-defined StructType (Scala) | StructType (Python) via DDL DataFrame command or JSON queried from AL schema repository.

```
%scala
import org.apache. spark.sql. types. {DataType, StructType}

val ddl = """alarm_status STRING, battery_level BIGINT,c02_level BIGINT,cca2 STRING,
cca3 STRING, cn STRING, coordinates STRUCT<latitude: DOUBLE, longitude: DOUBLE>,
date STRING, device_id BIGINT, device_serial_number STRING, device_type STRING,
epoch_time_miliseconds BIGINT, humidity BIGINT, ip STRING, scale STRING,temp DOUBLE,
timestamp STRING, _rescued_data STRING"""

val ddlSchema = StructType.fromDDL(ddl)

val json = """{"type":"struct","fields":[{"name":"alarm_status","type":"string",
"nullable":true,"metadata":{}},{"name":"battery_level","type":"long","nullable":
true,"metadata":{}},{"name":"c02_level","type":"long","nullable":true, "metadata"
:{}},{"name":"cca2","type":"string","nullable":true,"metadata":{}},{"name":"cca3",
"type":"string","nullable":true,"metadata":{}},{"name":"cn","type":"string","nullable":
true,"metadata":{}},{"name":"coordinates","type":{"type":"struct","fields":
[{"name":"latitude","type":"double","nullable":true,"metadata":{}},{"name":"longitude",
"type":"double","nullable":true,"metadata":{}}]},"nullable":true,"metadata":{}},{"name"
:"date","type":"string","nullable":true,"metadata":{}},{"name":"device_id","type":
"long","nullable":true,"metadata":{}},{"name":"deviceserial_number","type":"string",
"nullable":true,"metadata":{}},{"name":"device_type","type":"string","nullable":true,
"metadata":{}},{"name":"epochtime_miliseconds","type":"long","nullable":true,"metadata":{}}
,{"name":"humidity","type":"long","nullable":true,"metadata":{}},{"name":
"ip","type":"string","nullable":true,"metadata":{}},{"name":"scale","type":"string",
"nullable":true,"metadata":{}},{"name":"temp","type":"double","nullable":true,
"metadata":{}},{"name":"timestamp","type":"string","nullable":true,"metadata":{}}]}"""

val jsonSchema = DataType. fromJson(json).asInstanceOf[StructType]
```

```
%scala
val schemaAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.schema(jsonSchema) // schema structtype definition
.load(jsonSchema1Path)
)
```

```
%scala
schemaAlDf.printSchema
```

```
root
|-- alarm_status: string (nullable = true)
|-- battery_level: long (nullable = true)
|-- c02_level: long (nullable = true)
|-- cca2: string (nullable = true)
|-- cca3: string (nullable = true)
|-- cn: string (nullable = true)
|-- coordinates: struct (nullable = true)
| |-- latitude: double (nullable = true)
| |-- longitude: double (nullable = true)
|-- date: string (nullable = true)
|-- device_id: long (nullable = true)
|-- device_serial_number: string (nullable = true)
|-- device_type: string (nullable = true)
|-- epoch_time_miliseconds: long (nullable = true)
|-- humidity: long (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
```

Passing in the schema definition will enforce the stream. AL also provides a schema enforcement option achieving basically the same results as providing a static StructType schema-on-read. This method will be covered in Example 7.

```
%scala
display(schemaAlDf.limit(10))
```