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

EXAMPLE 5: SCHEMA DRIFT

AL stores new columns and data types via the rescue column. This column captures schema changes-on-read. The stream does not fail when schema and data type mismatches are discovered. This is a very impressive feature!

```scala
%scala
val driftAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.inferColumnTypes", true)
.option("cloudFiles.schemaEvolutionMode", "rescue") // schema drift tracking
.load(rawBasePath + "/*.json")
)
```

```scala
%scala
driftAlDf.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)
```

The rescue column preserves schema drift such as newly appended columns and/or different data types via a JSON string payload. This payload can be parsed via Spark DataFrame or Dataset APIs to analyze schema drift scenarios. The source file path for each individual row is also available in the rescue column to investigate the root cause.

```scala
%scala
display(driftAlDf.where("_rescued_data is not null").limit(10))
```

Table + New result table: OFF

	seconds	humidity	ip	scale	temp	timestamp	_rescued_data
1		null	213.161.254.1	Fahrenheit	51.8	2016-03-20 03:20:54.119	{"location":{"cca3":"NOR","cn":"Norway","cca2":"NO"},"latitude":62.47,"device_serial_number_device_type":"{\"2n2Pea\":\"sensor-pad\"}","longitude":6.15,"humidity":70.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
2		null	68.28.91.22	Fahrenheit	53.6	2016-03-20 03:20:54.126	{"location":{"cca3":"USA","cn":"United States","cca2":"US"},"latitude":38.0,"device_serial_number_device_type":"{\"12Y2klm0o\":\"sensor-pad\"}","longitude":-97.0,"humidity":92.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
3		null	59.90.65.1	Fahrenheit	73.4	2016-03-20 03:20:54.133	{"location":{"cca3":"IND","cn":"India","cca2":"IN"},"latitude":12.98,"device_serial_number_device_type":"{\"230IupA\":\"meter-gauge\"}","longitude":77.58,"humidity":47.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
4		null	213.7.14.1	Fahrenheit	75.2	2016-03-20 03:20:54.141	{"location":{"cca3":"CYP","cn":"Cyprus","cca2":"CY"},"latitude":35.0,"device_serial_number_device_type":"{\"36VQv8fnEg\":\"sensor-pad\"}","longitude":33.0,"humidity":47.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
5		null	62.128.16.74	Fahrenheit	80.6	2016-03-20 03:20:54.149	{"location":{"cca3":"DEU","cn":"Germany","cca2":"DE"},"latitude":49.46,"device_serial_number_device_type":"{\"448DeWGLL\":\"sensor-pad\"}","longitude":11.1,"humidity":63.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
6		null	170.37.224.1	Fahrenheit	77	2016-03-20 03:20:54.152	{"location":{"cca3":"USA","cn":"United States","cca2":"US"},"latitude":42.28,"device_serial_number_device_type":"{\"49YesGXwt\":\"meter-gauge\"}","longitude":-71.44,"humidity":70.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
7		null	38.99.198.186	Fahrenheit	53.6	2016-03-20 03:20:54.162	{"location":{"cca3":"USA","cn":"United States","cca2":"US"},"latitude":38.0,"device_serial_number_device_type":"{\"64djcln\":\"sensor-pad\"}","longitude":-97.0,"humidity":55.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
8		null	218.248.255.30	Fahrenheit	62.6	2016-03-20 03:20:54.169	{"location":{"cca3":"IND","cn":"India","cca2":"IN"},"latitude":12.98,"device_serial_number_device_type":"{\"77IKW3YAB55\":\"meter-gauge\"}","longitude":77.58,"humidity":82.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
9		null	159.128.0.181	Fahrenheit	89.6	2016-03-20 03:20:54.171	{"location":{"cca3":"CAN","cn":"Canada","cca2":"CA"},"latitude":50.01,"device_serial_number_device_type":"{\"80TY4dWSMH\":\"sensor-pad\"}","longitude":-97.22,"humidity":57.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}
10		null	24.32.26.1	Fahrenheit	82.4	2016-03-20 03:20:54.179	{"location":{"cca3":"USA","cn":"United States","cca2":"US"},"latitude":39.33,"device_serial_number_device_type":"{\"94HL9jChDl\":\"sensor-pad\"}","longitude":-120.25,"humidity":66.0,"_file_path":"dbfs:/user/garrett.peternel@databricks.com/raw/iot-schema-2.json/iot_schema_2.json"}

EXAMPLE 6: SCHEMA EVOLUTION

AL merges schemas as new columns arrive via schema evolution mode. New schema JSON will be updated and stored as a new version in the specified schema repository location.

```scala
%scala
val evolveAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.inferColumnTypes", true)
.option("cloudFiles.schemaEvolutionMode", "addNewColumns") // schema evolution
.load(rawBasePath + "/*.json")
)
```

```scala
%scala
evolveAlDf.printSchema // original schema
```

```
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)
```

```scala
%scala
display(evolveAlDf.limit(10)) // # stream will fail
```

AL purposely fails the stream with an UnknownFieldException error when it detects a schema change via dynamic schema inference. The updated schema instance is created as a new version and metadata file in the schema repository location, and will be used against the input data after restarting the stream.

```scala
%scala
val evolveAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.inferColumnTypes", true)
.option("cloudFiles.schemaHints", "humidity DOUBLE")
.option("cloudFiles.schemaEvolutionMode", "addNewColumns") // schema evolution
.load(rawBasePath + "/*.json")
)
```

```scala
%scala
evolveAlDf.printSchema // evolved schema
```

```
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: double (nullable = true)
|-- ip: string (nullable = true)
|-- scale: string (nullable = true)
|-- temp: double (nullable = true)
|-- timestamp: string (nullable = true)
|-- device_serial_number_device_type: 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)
|-- _rescued_data: string (nullable = true)
```

AL has evolved the schema to merge the newly acquired data fields.

```scala
%scala
display(evolveAlDf.where("device_serial_number_device_type is not null").limit(10))
```

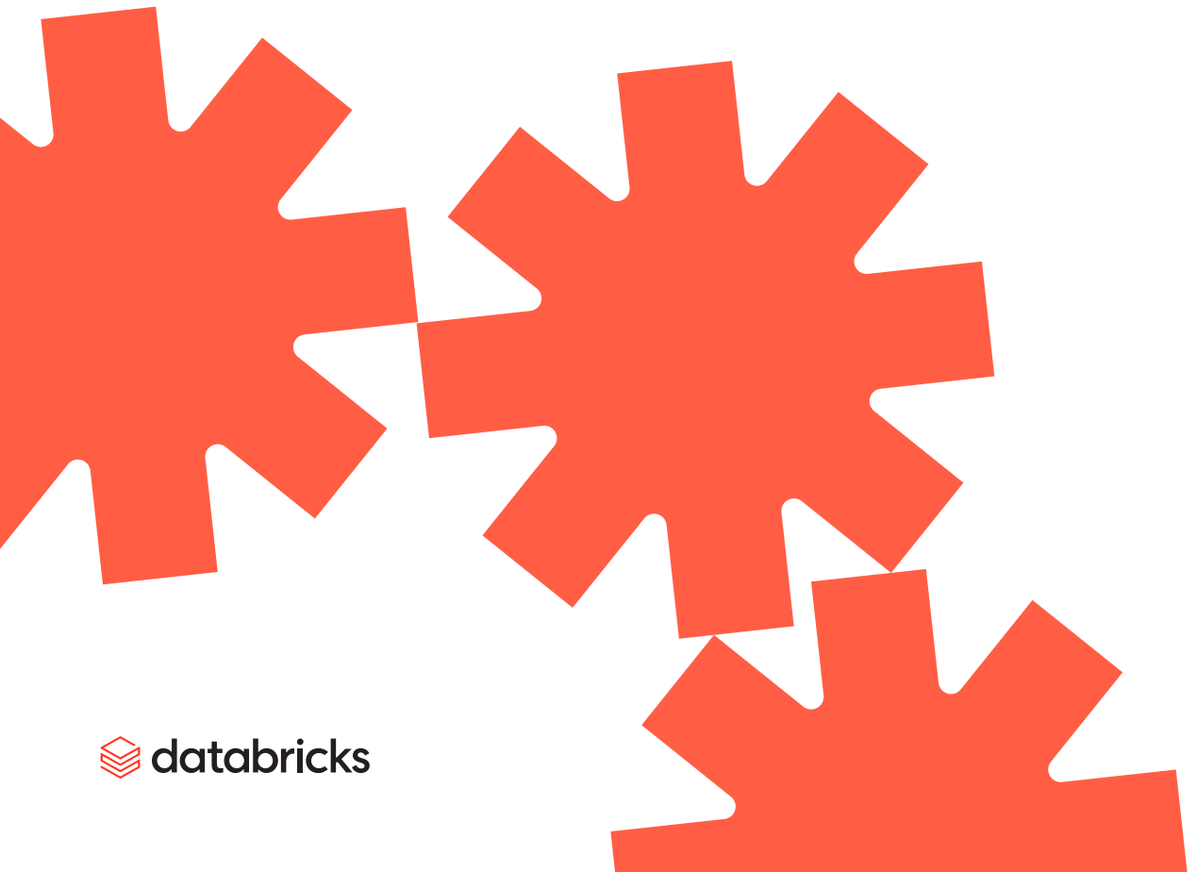


Table + New result table: OFF

	temp	timestamp	device_serial_number_device_type	latitude	location	longitude	_rescued_data
1	51.8	2016-03-20 03:20:54.119	{"2n2Pea":"sensor-pad"}	62.47	{"cca2": "NO", "cca3": "NOR", "cn": "Norway"}	6.15	null
2	53.6	2016-03-20 03:20:54.126	{"12Y2klm0o":"sensor-pad"}	38	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-97	null
3	73.4	2016-03-20 03:20:54.133	{"230IupA":"meter-gauge"}	12.98	{"cca2": "IN", "cca3": "IND", "cn": "India"}	77.58	null
4	75.2	2016-03-20 03:20:54.141	{"36VQv8fnEg":"sensor-pad"}	35	{"cca2": "CY", "cca3": "CYP", "cn": "Cyprus"}	33	null
5	80.6	2016-03-20 03:20:54.149	{"448DeWGL":"sensor-pad"}	49.46	{"cca2": "DE", "cca3": "DEU", "cn": "Germany"}	11.1	null
6	77	2016-03-20 03:20:54.152	{"49YesGXwt":"meter-gauge"}	42.28	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-71.44	null
7	53.6	2016-03-20 03:20:54.162	{"64djcln":"sensor-pad"}	38	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-97	null
8	62.6	2016-03-20 03:20:54.169	{"77IKW3YAB55":"meter-gauge"}	12.98	{"cca2": "IN", "cca3": "IND", "cn": "India"}	77.58	null
9	89.6	2016-03-20 03:20:54.171	{"80TY4dWSMH":"sensor-pad"}	50.01	{"cca2": "CA", "cca3": "CAN", "cn": "Canada"}	-97.22	null
10	82.4	2016-03-20 03:20:54.179	{"94HL9jChD":"sensor-pad"}	39.33	{"cca2": "US", "cca3": "USA", "cn": "United States"}	-120.25	null

The newly merged schema transformed by AL is stored in the original schema repository path as version 1 along with the base version 0 schema. This history is valuable for tracking changes to schema over time, as well as quickly retrieving DDL on the fly for schema enforcement.

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	1709325391000
2	dbfs:/user/garrett.peternel@databricks.com/repo/iot-ddl.json/_schemas/1	1	2209	1709325578000

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":[]}
3	v1	null	null
4	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":[]}

Schema evolution can be a messy problem if frequent. With AL and Delta Lake it becomes easier and simpler to manage. Adding new columns is relatively straightforward as AL combined with Delta Lake uses schema evolution to append them to the existing schema. Note: the values for these columns will be NULL for data already processed. The greater challenge occurs when the data types change because there will be a type mismatch against the data already processed. Currently, the "safest" approach is to perform a complete overwrite of the target Delta table to refresh all data with the changed data type(s). Depending on the data volume this operation is also relatively straightforward if infrequent. However, if data types are changing daily/weekly then this operation is going to be very costly to reprocess large data volumes. This can be an indication that the business needs to improve their data strategy.

Constantly changing schemas can be a sign of a weak data governance strategy and lack of communication with the data business owners. Ideally, organizations should have some kind of SLA for data acquisition and know the expected schema. Raw data stored in the landing zone should also follow some kind of pre-ETL strategy (e.g., ontology, taxonomy, partitioning) for better incremental loading performance into the lakehouse. Skipping these steps can cause a plethora of data management issues that will negatively impact downstream consumers building data analytics, BI, and AI/ML pipelines and applications. If upstream schema and formatting issues are never addressed, downstream pipelines will consistently break and result in increased cloud storage and compute costs. Garbage in, garbage out.

EXAMPLE 7: SCHEMA ENFORCEMENT

AL validates data against the linked schema version stored in repository location via schema enforcement mode. Schema enforcement is a schema-on-write operation, and only ingested data matching the target Delta Lake schema will be written to output. Any future input schema changes will be ignored, and AL streams will continue working without failure. Schema enforcement is a very powerful feature of AL and Delta Lake. It ensures only clean and trusted data will be inserted into downstream Silver/Gold datasets used for data analytics, BI, and AI/ML pipelines and applications.

```
%scala
val enforceAlDf = (spark
.readStream.format("cloudfiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.schemaLocation", repoSchemaPath)
.option("cloudFiles.schemaEvolutionMode", "none") // schema enforcement
.schema(jsonSchema)
.load(rawBasePath + "/*.json")
)
```

```
%scala
enforceAlDf.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)
```

Please note the rescue column is no longer available in this example because schema enforcement has been enabled. However, a rescue column can still be configured separately as an AL option if desired. In addition, schema enforcement mode uses the latest schema version in the repository to enforce incoming data. For older versions, set a user-defined schema as explained in Example 4.

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

Table +

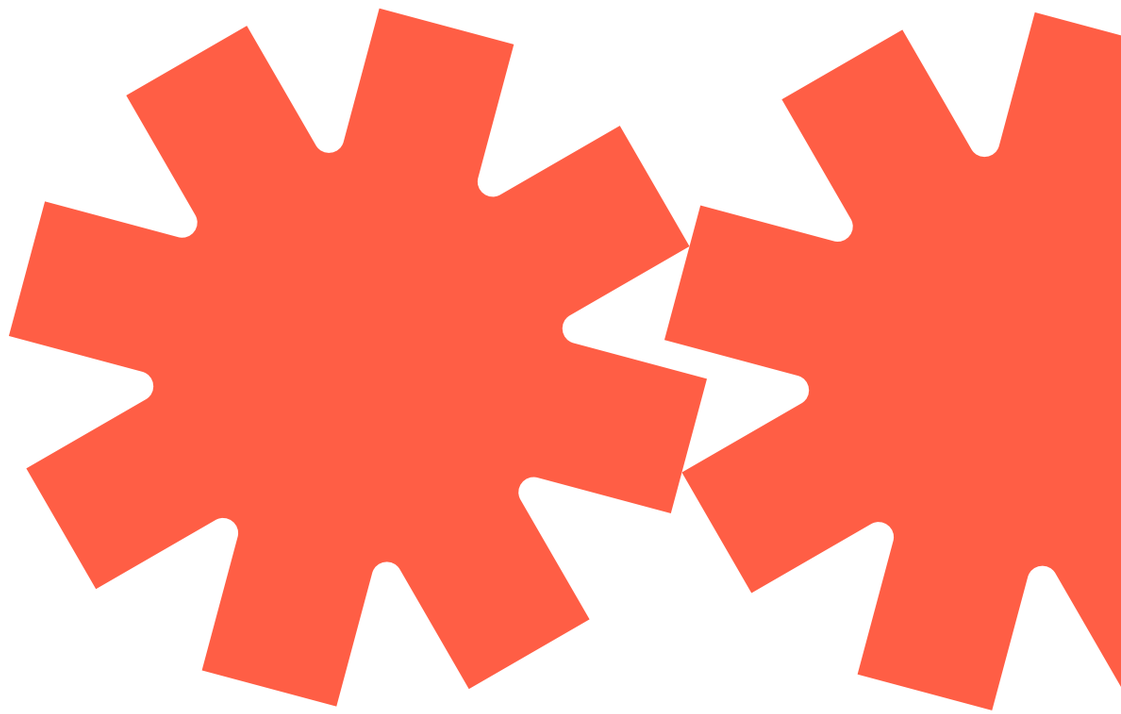
New result table: OFF

	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

CONCLUSION

At the end of the day, data issues are inevitable. However, the key is to limit data pollution as much as possible and have methods to detect discrepancies, changes and history via schema management. Databricks Auto Loader provides many solutions for schema management, as illustrated by the examples in this chapter. Having a solidified data governance and landing zone strategy will make ingestion and streaming easier and more efficient for loading data into the lakehouse. Whether it is simply converting raw JSON data incrementally to the Bronze layer as Delta Lake format, or having a repository to store schema metadata, AL makes your job easier. It acts as an anchor to building a resilient lakehouse architecture that provides reusable, consistent, reliable and performant data throughout the data and AI lifecycle.

HTML notebooks (Spark Scala and Spark Python) with code and both sample datasets can be found at the GitHub repo here.



From Idea to Code: Building With the Databricks SDK for Python

by Kimberly Mahoney

The focus of this chapter is to demystify the Databricks SDK for Python — the authentication process and the components of the SDK — by walking through the start-to-end development process. I'll also be showing how to utilize IntelliSense and the debugger for real-time suggestions in order to reduce the amount of context-switching from the IDE to documentation and code examples.

What is the Databricks SDK for Python . . . and why you should use it

The Databricks Python SDK lets you interact with the Databricks Platform programmatically using Python. It covers the entire Databricks API surface and Databricks REST operations. While you can interact directly with the API via curl or a library like 'requests' there are benefits to utilizing the SDKs such as:

- Secure and simplified authentication via Databricks client-unified authentication
- Built-in debug logging with sensitive information automatically redacted
- Support to wait for long-running operations to finish (kicking off a job, starting a cluster)
- Standard iterators for paginated APIs (we have multiple pagination types in our APIs!)
- Retrying on transient errors

There are numerous practical applications, such as building multi-tenant web applications that interact with your ML models or a robust UC migration toolkit like Databricks Labs project UCX. Don't forget the silent workhorses — those simple utility scripts that are more limited in scope but automate an annoying task such as bulk updating cluster policies, dynamically adding users to groups or simply writing data files to UC Volumes. Implementing these types of scripts is a great way to familiarize yourself with the Python SDK and Databricks APIs.

SCENARIO

Imagine my business is establishing best practices for development and CI/CD on Databricks. We're adopting DABs to help us define and deploy workflows in our development and production workspaces, but in the meantime, we need to audit and clean up our current environments. We have a lot of jobs people created in our dev workspace via the UI. One of the platform admins observed many of these jobs are inadvertently configured to run on a recurring schedule, racking up unintended costs. As part of the cleanup process, we want to identify any scheduled jobs in our development workspace with an option to pause them. We'll need to figure out:

- How to install the SDK
- How to connect to the Databricks workspace
- How to list all the jobs and examine their attributes
- How to log the problematic jobs — or a step further, how to call the API to pause their schedule

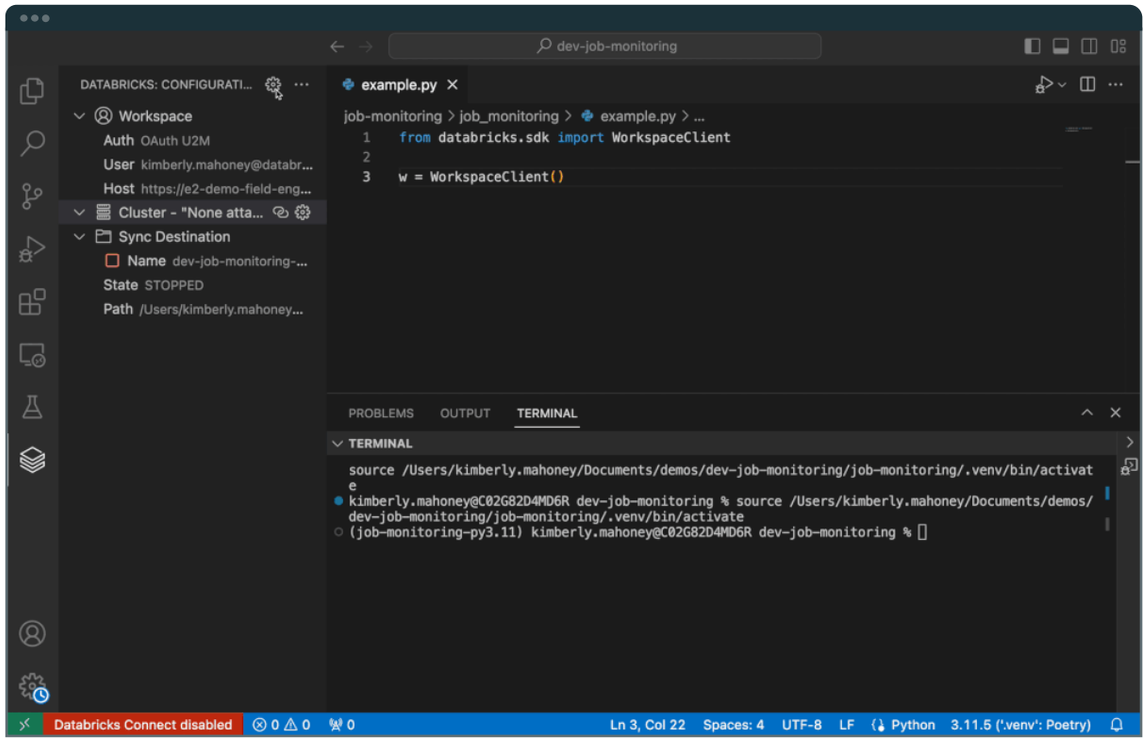
DEVELOPMENT ENVIRONMENT

Before diving into the code, you need to set up your development environment. I highly recommend using an IDE that has a comprehensive code completion feature as well as a debugger. Code completion features, such as IntelliSense in VS Code, are really helpful when learning new libraries or APIs — they provide useful contextual information, autocompletion, and aid in code navigation. For this chapter, I'll be using Visual Studio Code so I can also make use of the Databricks Extension as well as Pylance. You'll also need to install the databricks-sdk (docs). In this chapter, I'm using Poetry + Pyenv. The setup is similar for other tools — just 'poetry add databricks-sdk' or alternatively 'pip install databricks-sdk' in your environment.

AUTHENTICATION

The next step is to authorize access to Databricks so we can work with our workspace. There are several ways to do this, but because I'm using the VS Code Extension, I'll take advantage of its authentication integration. It's one of the tools that uses unified client authentication — that just means all these development tools follow the same process and standards for authentication and if you set up auth for one, you can reuse it among the other tools. I set up both the CLI and VS Code Extension previously, but here is a primer on setting up the CLI and installing the extension. Once you've connected successfully, you'll see a notification banner in the lower right-hand corner and see two hidden files generated in the *.databricks* folder — project.json and databricks.env (don't worry, the extension also handles adding these to .gitignore).

For this example, while we're interactively developing in our IDE, we'll be using what's called U2M (user-to-machine) OAuth. We won't get into the technical details, but OAuth is a secure protocol that handles authorization to resources without passing sensitive user credentials such as PAT or username/password that persist much longer than the one-hour short-lived OAuth token.



OAuth flow for the Databricks Python SDK

WORKSPACECLIENT VS. ACCOUNTCLIENT

The Databricks API is split into two primary categories — account and workspace. They let you manage different parts of Databricks, like user access at the account level or cluster policies in a workspace. The SDK reflects this with two clients that act as our entry points to the SDK — the WorkspaceClient and AccountClient. For our example we'll be working at the workspace level, so I'll be initializing the WorkspaceClient. If you're unsure which client to use, check out the SDK documentation.

Tip: Because we ran the previous steps to authorize access via unified client auth, the SDK will automatically use the necessary Databricks environment variables, so there's no need for extra configurations when setting up your client. All we need are these two lines of code:

- *Initializing our WorkspaceClient*

```
from databricks.sdk import WorkspaceClient

w = WorkspaceClient()
```

MAKING API CALLS AND INTERACTING WITH DATA

The WorkspaceClient we instantiated will allow us to interact with different APIs across the Databricks workspace services. A **service** is a smaller component of the Databricks Platform — e.g., Jobs, Compute, Model Registry.
In our example, we'll need to call the Jobs API in order to retrieve a list of all the jobs in the workspace.

```
import logging
import sys

w = WorkspaceClient()

w.
```

```
git_credentials
global_init_scripts
grants
groups
instance_pools
instance_profiles
ip_access_lists
jobs
libraries
metastores
model_registry
model_versions
```

Services accessible via the Python SDK