Orchestration with Airflow Data wrangling Deployment

Instructor: Dr. GP Saggese - gsaggese@umd.edu**

v1.1

UMD DATA605 - Big Data Systems

UMD DATA605 - Big Data Systems Orchestration with Airflow

UMD DATA605 - Big Data Systems Orchestration with Airflow Data wrangling Deployment

Dr. GP Saggese gsaggese@umd.edu

Serialization Formats

- Programs need to send data to each other (on the network, on disk)
  - E.g., Remote Procedure Calls (RPCs)
  - Several recent technologies based around schemas
    - JSON, YAML, Protocol Buffer, Python Pickle
- Serialization formats are data models

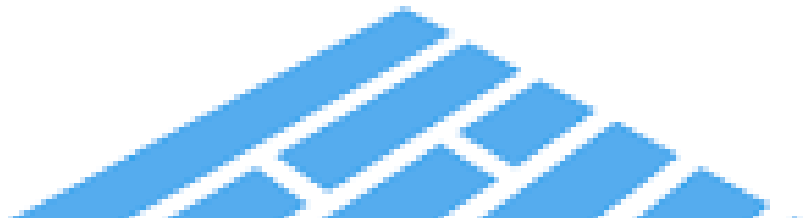
Comma Separated Values (CSV)

- CSV stores data row-wise as text without schema
  - Each line of the file is a data record
  - Each record consists of one or more fields, separated by commas
- **Pros**
  - Very portable
    - It's text
    - Supported by every tool
  - Human-friendly
- **Cons**
  - Large footprint
    - Compression
  - Parsing is CPU intensive
  - No easy random access
  - No read only a subset of columns
  - No schema / types
    - Annotate CSV files with schema
  - Mainly read-only, difficult to modify

Year	Make	Model	Description	Price
1997	Ford	E350	ac, abs, moon	3000.00
1999	Chevy	Venture "Extended Edition"		4900.00

(Apache) Parquet

- Parquet allows to read tiles of data
  - That's what the name comes from
- Supports multi-dimensional and nested data
  - A generalization of dataframes
- Column-storage
  - Each column is stored together, has uniform data type, and compressed (efficiently)
- Queries can be executed by IO layer
  - Only the necessary chunks of data is read from disk
- **Pros**
  - 10x smaller than CSV
  - 10x faster (with multi-threading)
  - You can read only a subset of columns and rows
- **Cons**
  - Binary, non-human friendly
  - Need ingestion step converting the inbound format to Parquet
  - Mainly read-only, difficult to modify



JSON

- JSON = JavaScript Object Notation
- Data is nested dictionaries and arrays
- Very similar to XML
  - More human-readable
  - Less boilerplate
  - Executable in JavaScript (and Python)

```
{     "firstName": "John",     "lastName": "Smith",     "isAlive":
   "phoneNumbers": [         {         "type": "home",         "numbe
```

Protocol Buffers

- Developed by Google
- Open-source
- Represent data structures in:
  - Language agnostic
  - Platform agnostic
  - Versioning
- Schema is mostly relational
  - Optional fields
  - Types
  - Default values
  - Structures
  - Arrays
- Schema specified using a **.proto** file
- Compiled by **protoc** to produce C++, Java, or Python code to initialize, read, serialize objects

```
import addressbook_pb2
person = addressbook_pb2.Person()
person.id = 1234
person.name = "John Doe"
person.email = "jdoe@example.com"
```

Serialization Formats

- Avro
  - Richer data structures
  - JSON-specified schema
- Thrift
  - Developed by Facebook
  - Now Apache project
  - More languages supported
  - Supports exceptions and sets

```
{       "namespace": "example.avro",        "type": "record",
```

Remote Procedure Call

- **Remote Procedure Call** (RPC) is a protocol to request a service from a program located in another computer abstracting the details of the network communication
- **Goal**: similar to how procedure calls are made within a single process, without having to understand the network's details
- **Problems**
  - Can't serialize pointers
  - Asynchronous communication
  - Failures and retry
- Used in distributed systems
  - E.g., microservices architectures, cloud services, and client-server applications
- Can be synchronous or asynchronous

RPCs: Internals

- **Client procedure call**: Client calls a stub function, providing the necessary arguments
- **Request marshalling**: Client stub serializes the procedure's arguments into a format suitable for transmission over the network
- **Server communication**: Client's RPC runtime sends the procedure request across the network to the server
- **Server-side unmarshalling**: Server's RPC runtime receives the request and deserializes the arguments
- **Procedure execution**: Server calls the actual procedure on the server-side
- **Response marshalling**: Once the procedure completes, the return values are marshaled into a response message
- **Client communication** / **response unmarshalling** / **return to client**: Return values are passed back to the client's original stub call, and execution continues as if the call were local.