COSC 6339
Big Data Analytics

File Formats (III)
avro and protocol buffers

3rd Homework assignment

Edgar Gabriel
Spring 2020

Avro

- Language-neutral data serialization system
  - Serialization: process of translating data structures or objects state into binary format
- Schema-based system
- Creates binary structured format that is both compressible and splittable
- Schemas defined in JSON

- Avro follows its own standards of defining schemas.
- Schemas describe by
  - **type**: Describes document type, in this case a "record".
  - **namespace:** Describes the name of the namespace in which the object resides.
  - **name**: Describes the schema name.
  - **fields**: This is an attribute array which contains the following
    - **name:** Describes the name of field
    - **type**: Describes data type of field

Example Avro schema

```
{
   "type" : "record",
   "namespace" : "COSC6339",
   "name" : "Student",
   "fields" : [
      { "name" : "Name" , "type" : "string" },
      { "name" : "StudentID" , "type" : "int" }
   ]
}
```

Primitive Avro Data types

Data type	Description
null	Null is a type having no value.
int	32-bit signed integer.
long	64-bit signed integer.
float	single precision (32-bit) IEEE 754 floating-point number.
double	double precision (64-bit) IEEE 754 floating-point number.
bytes	sequence of 8-bit unsigned bytes.
string	Unicode character sequence.

Complex Avro Data Types

- Record: a collection of multiple attributes.
- Enum: a list of enumerated items in a collection
- Arrays: array field having a same attribute(s) for all elements
- Maps: array of key-value pairs
- Unions: used whenever a field has one or more datatypes

Reading/Writing Avro files

- Option 1: use the Avro code generator to create a specific class to read an Avro file given the schema, .e.g in Java

```
gabriel@whale> cat pair.avsc
{"namespace" : "example.avro",
 "type": "record",
 "name": "Pair",
 "doc": "A pair of strings.",
 "fields": [
            {"name": "first", "type": "string"},
            {"name": "second", "type": "string"}
           ]
}
gabriel@whale> java -jar avro-tools-1.8.2.jar compile schema
pair.avsc /home/hadoopsys/tmp/avro-test/jars/
```

Reading/Writing Avro files

- For C++:

```
avrogencpp -i pair.avsc -o pair.h -n mynamespace
```

  will generate a file called `pair.h` that can be included and used in a C++ file.

- Option 2: use the Avro parser library
  - data is always stored with its corresponding schema

Avro in Python

- Avro Python library does not support code generation for schema.
- Need to parse the schema at the time of writing avro data file itself

```
import avro.schema
from avro.datafile import DataFileReader, DataFileWriter
from avro.io import DatumReader, DatumWriter

# Schema parsing from a schema file
schema = avro.schema.parse(open("pair.avsc").read())

# Creation of DataFileWriter instance with above schema
writer = DataFileWriter(open("pairs.avro", "w"), DatumWriter(), schema)

# Write some pair records
writer.append({"first": "left", "second": "right"})
writer.append({"first": "correct", "second": "wrong"})
writer.append({"first": "good", "second": "bad"})
# Close the data file
writer.close()
```

Reading & writing Avro files in pyspark

Based on the spark-avro package

For Spark DataFrames

```
df = sqlContext.read.format("com.databricks.spark.avro").
load( "/gabriel/users.avro")

df.write.format("com.databricks.spark.avro").save("/gabrie
l/output.avro")
```

For RDDs:

```
rdd =  spark.read.format("com.databricks.spark.avro").
load("/gabriel/users.avro")

rdd.write.format("com.databricks.spark.avro").save("/gabri
el/output.avro")
```

Protocol Buffers (protobuf)

- Binary serialization format developed by Google
- Language independent
- Data structures (*messages*) described in a proto definition file (.proto)
- Source code generated using the `protoc` compiler

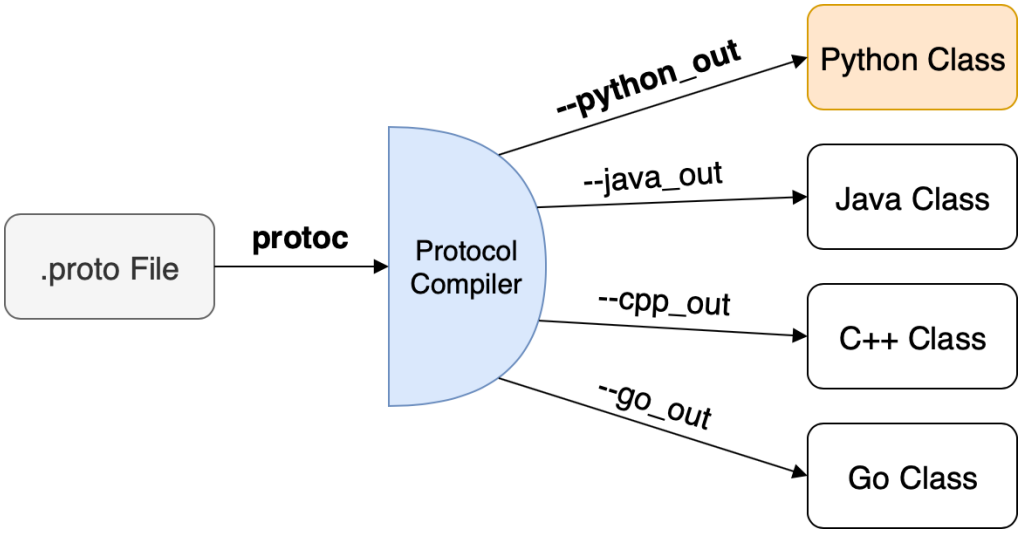


Image source: https://www.freecodecamp.org/news/googles-protocol-buffers-in-python/

Protocol Buffers

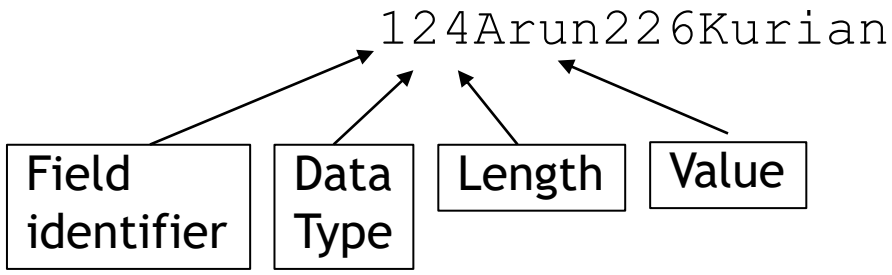
- Separation of context and data
  - In contrary to XML and Json
- Example in json:

```
{
        first_name: "Arun",
        last_name: "Kurian"
}
```

- In Protobuf:

```
{
        string first_name =1;
        string last_name = 2;
}
```

- Encoded data



Slide based on example shown at:
https://medium.com/better-programming/understanding-protocol-buffers-43c5bced0d47

Protocol Buffers

- Protobuf specification changed over time
  - Current: version 3
- Field name rules
  - All field names need to be lower case
  - No spaces in the names, use underscore instead
- Field tag rules
  - Numerical (integer) value
  - Value must be unique inside a message
  - If a field is removed from the specification at a later stage, need to block the field tag out for future usage, e.g.

```
reserverd 8;
```

Protocol Buffers

- Arrays can be defined using the `repeated` field, e.g.

```
repeated int32 data = 4 [packed=true];
```

- `packed=true`
  - no need to add header before every value
  - Default since protobuf v3

Serialization formats

- Lots of other formats still available
  - Thrift: similar to protobuf but from facebook
  - Bond: similar to protobuf but from Microsoft
  - FlatBuffers: similar to protobuf, also from Google, but avoids a memcpy operations during serialization
  - Bson: Binary Json
  - MessagePack: binary serialization format, similar to Bson
  - And many, many, more…
- Challenging to support many/most of them because of the diversity of available solutions
  - E.g. hard to figure out what is supported e.g. by Spark!

3rd Homework Assignment

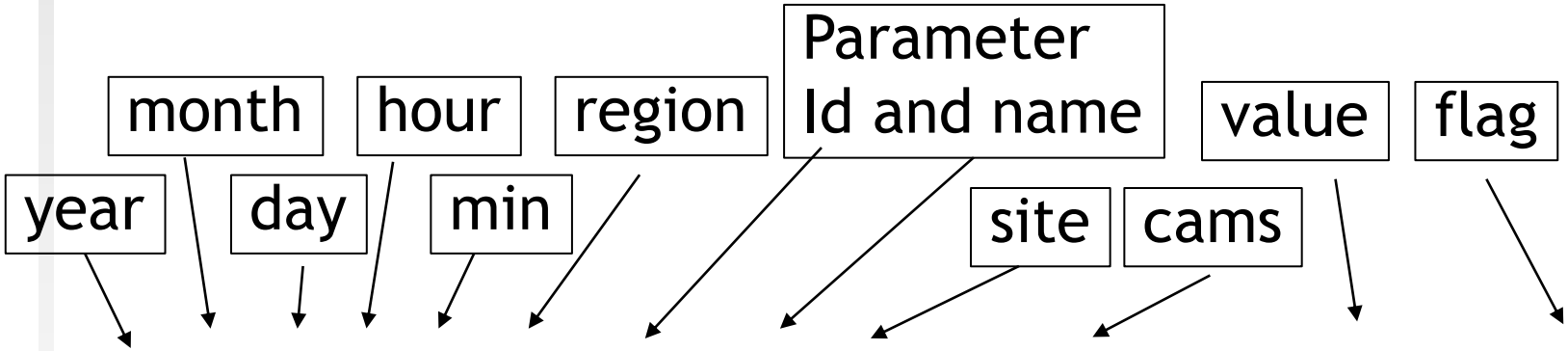
- Rules
  - Each student should deliver
    - Source code (.py files) compressed to a zip or tar.gz file
    - Documentation (.pdf, .docx, or .txt file)
      - explanations to the code
      - answers to questions
  - Deliver electronically on **blackboard**
  - Expected by Wednesday, November 11, 11.59pm
  - In case of questions: ask early!

- Given a data set containing measurements of pollutants from a large number sensors for the year 2013 in the state of Texas
  - each line is one data point with information as listed on the next page

- **Part 1:** Develop a pyspark code that calculates hourly average of O3 concentration **for each site separately**.
  - Only data points that have an empty flag can be used for calculation (Flags often indicate problems, e.g. calibration etc, and data is useless if flag is set)
  - If an hourly average can not be calculated, provide a constant (-1, NaN, NULL etc)

  - **Measure the execution time using** 1,2,4 and 8 executors using 2 cores per executor for the `2013_data_jan.csv` (~780MB) and `2013_data_full.csv` (~9GB)

File Format




```
2013,01,01,00,25,10,42101,"co",628,"C628_42101_1",152.88725,""
2013,01,01,00,25,10,42101,"co",1035,"C1035_42101_1",231.42252,""
2013,01,01,00,30,10,42101,"co",628,"C628_42101_1",196.76671,""
2013,01,01,00,30,10,42101,"co",1035,"C1035_42101_1",225.64098,""
2013,01,01,00,00,10,44201,"o3",2,"C2_44201_2",-8.09507,""
2013,01,01,00,00,10,44201,"o3",9,"C9_44201_2",25.60723,""
2013,01,01,00,00,1,61103,"windspd",79,"C79_61103_1",0.3,"PMA"
2013,01,01,00,00,1,61103,"windspd",104,"C104_61103_1",0.3,"LIM"
…
```

- Part 2: develop pyspark code to convert the csv file into two other formats, (hdf5, parquet, json, xml, avro)
  - You can also suggest another format. If you would like to try another format, drop me an email and if I approve it you can use it instead of one of the formats listed above.

Compare the file size of the new formats with the original csv file for 2013_data_full.csv

- Part 3: create variants of the code that you developed in part 1 using the two file formats that you chose in part 2.

Compare the execution time for the full data set using the three different formats (csv + the 2 formats that you chose in part 2) using 1, 2, 4, and 8 executors.

Revamped utilization of spark on the crill cluster

- Not using `salloc` and `crill-spark-submit` for this assignment anymore.
- Use instead directly `spark-submit`

```
spark-submit --master spark://crill:28959
    --total-executor-cores 8 --executor-cores 2
    ./mycode.py
    /home2/input/2013_data_tiny.csv
    /home2/output/stud58/testdir
```

- Input data available for all students in

  `/home2/input/`

- Results of spark job **must** be written to

  `/home2/output/stud<xy>`

  each student has a separate directory in the output folder. Replace `stud<xy>` with your own id (e.g. `stud58`)

- After running a Spark job, the resulting output files are owned by the user `spark`, not by `stud<xy>`
- Need to change ownership using a special script

```
sudo /opt/spark/2.3.4/bin/changeowner.sh stud<xy>
```

Command will prompt you for your password on crill

- Manipulating the number of executors:
  - Keep `executor-cores` always at 2
  - `total-executor-cores = executor-cores * number of executors` that you want to use,

  e.g. for 2 executors

```
spark-submit --master spark://crill:28959
   --total-executor-cores 4 --executor-cores 2
   ./mycode.py /home2/input/2013_data_tiny.csv
   /home2/output/stud58/testdir
```

  e.g. for 4 executors

```
spark-submit --master spark://crill:28959
   --total-executor-cores 8 --executor-cores 2
   ./mycode.py /home2/input/2013_data_tiny.csv
   /home2/output/stud58/testdir2
```

Documentation

- The Documentation should contain
  - (Brief) Problem description
  - Solution strategy
  - Description of how to run your code
  - Results section
    - Description of resources used
    - Description of measurements performed
    - Results (graphs/tables + findings)

- The document should **not** contain
  - Replication of the entire source code – that's why you have to deliver the sources
  - Screen shots of every single measurement you made
    - Actually, no screen shots at all.
  - The output files