Data Ingestion

Application in VerticaPy

20 mins

Table of Contents

In this lesson we will try to import different datasets and get familiar with the format and nuances of data ingestion.

What you will learn in this module?

- What does materialize mean?
- How to import single files?

- How to import complex data?
- What are the formats currently accepted in VerticaPy?
- How to import a batch of files?

Data Ingestion in Vertica

As you may know, VerticaPy is the python API for Vertica. So, almost all the calcualtion/computations are actually performed inside Vertica servers. This means that during meaningful work, the data first has to get transfered to Vertica from your local computer or your data repository where you want to extract the data from. The process of creating a copy of data inside Vertica is called "**materialize**".

Altogether, we have three options when we are ingesting data:

- **Materialize:** A copy of the data is created inside the Vertica server which will remain there until deleted.
- **Create Temp Table:** A copy of the data/table is created on the server which will be deleted after a certain time, but not during the session.
- **Create local temporary table (in-memory):** A copy of table is created inside the local random access memory of the user/client machine. This table will be deleted at the end of the session.

With the above in mind, let us begin ingesting data.

Comma Separated Variables (CSV) Files

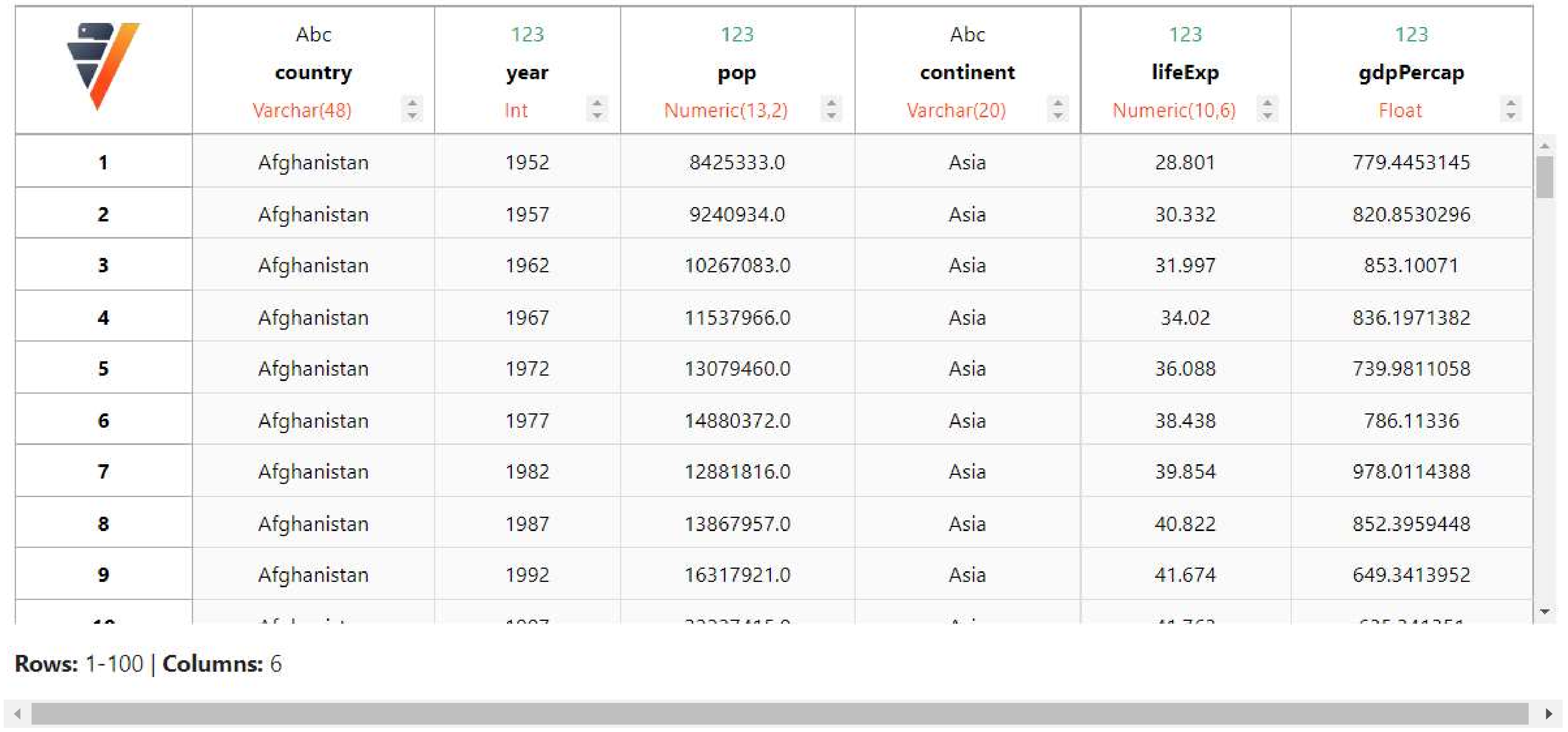
CSV is perhaps the most common type of data format for simple structured data. We can now use VerticaPy to ingest CSV data and explore the options that we have.

For example, we have a csv file "gapminder.csv" which has temporal development data concerning some countries across the world. Since it is a CSV file, we can use the VerticaPy read_csv function.

```python
import verticapy as vp
data=vp.read_csv("gapminder.csv")
```

The above code will ingest the data and create a vDataFrame containing all the data. let's look at the data:

```
data
```



	Abc **country** Varchar(48)	123 **year** Int	123 **pop** Numeric(13,2)	Abc **continent** Varchar(20)	123 **lifeExp** Numeric(10,6)	123 **gdpPercap** Float
1	Afghanistan	1952	8425333.0	Asia	28.801	779.4453145
2	Afghanistan	1957	9240934.0	Asia	30.332	820.8530296
3	Afghanistan	1962	10267083.0	Asia	31.997	853.10071
4	Afghanistan	1967	11537966.0	Asia	34.02	836.1971382
5	Afghanistan	1972	13079460.0	Asia	36.088	739.9811058
6	Afghanistan	1977	14880372.0	Asia	38.438	786.11336
7	Afghanistan	1982	12881816.0	Asia	39.854	978.0114388
8	Afghanistan	1987	13867957.0	Asia	40.822	852.3959448
9	Afghanistan	1992	16317921.0	Asia	41.674	649.3413952

Rows: 1-100 | **Columns:** 6

Note how VerticaPy automatically detected the data types for each of the columns in our table.

Using the previously mentioned syntax, we only ingested the data locally i.e., we only created a temporary table. but if we wanted to store the table in our Vertica server than we can easily do this by switching on the **materialize** parameter and providing the schema and table name:

```
data=vp.read_csv("gapminder.csv",
                 table_name = "gapminder_data",
                 schema = "public",
                 materialize = True
                )
```

Another parameter inside read_csv function is **omit_empty_keys**. This parameters allows the user to ignore the columns whose headers are missing. VerticaPy also has many other amazing options inside read_csv function which are encouraged to check out in detail here..

JSON Files and complex data

As mentioned in previous lesson, JSON files are very efficient in storing nested complex data. Let us ingest data of a soccer tournaments for year 2005 using the read_json function:

```
data=vp.read_json("2005.json")
```

[8]:

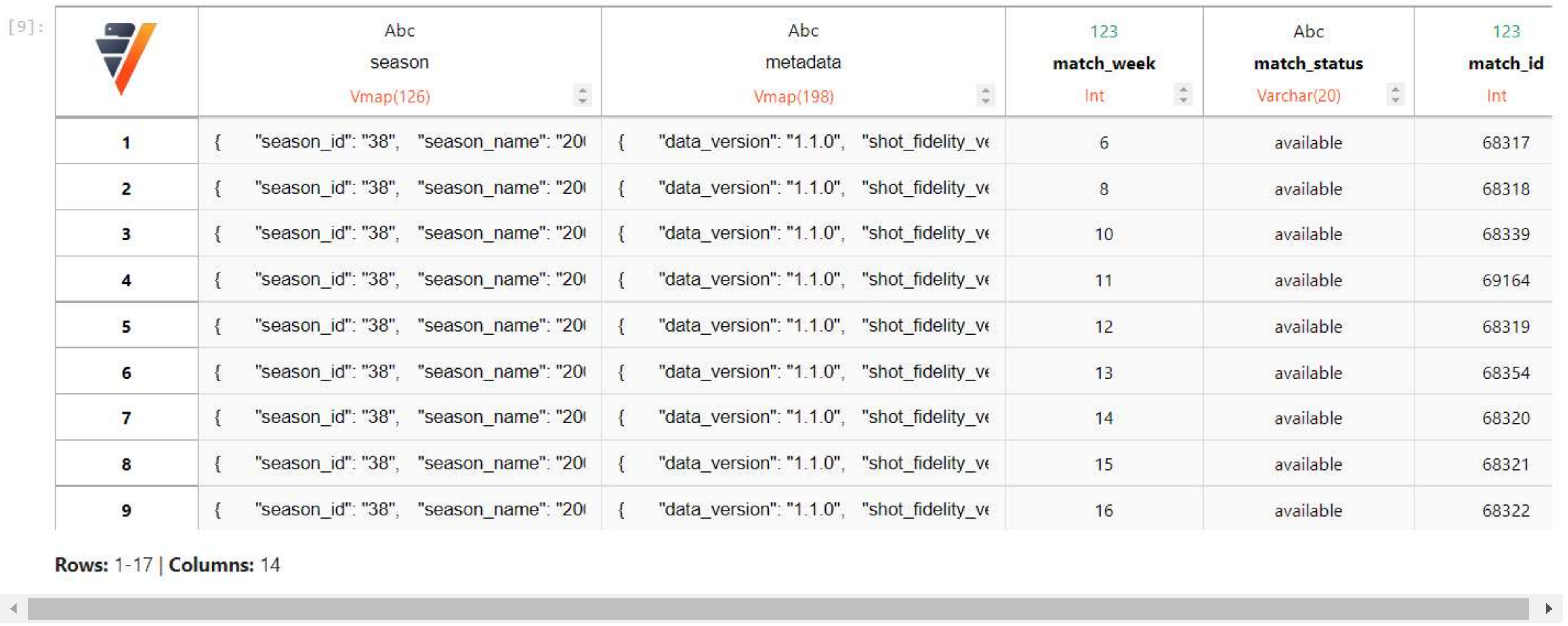
	Abc **season.season_name** Varchar(20)	123 **season.season_id** Int	123 **metadata.xy_fidelity_version** Int	123 **metadata.shot_fidelity_version** Int	**metadata.data_vers** Date
1	2005/2006	38	2	2	2000-01-01
2	2005/2006	38	2	2	2000-01-01
3	2005/2006	38	2	2	2000-01-01
4	2005/2006	38	2	2	2000-01-01
5	2005/2006	38	2	2	2000-01-01
6	2005/2006	38	2	2	2000-01-01
7	2005/2006	38	2	2	2000-01-01
8	2005/2006	38	2	2	2000-01-01
9	2005/2006	38	2	2	2000-01-01

Rows: 1-17 | **Columns:** 30

When we look at the resulting table, we see that there are 30 columns. VerticaPy has automatically flattened the structure of the nested table. This is because the defaul value of the **flatten_maps** parameter is set to True.

In order to keep the originla structure, we can turn this value to False and get the nested structure as output.

```python
data=vp.read_json("2005.json",
                  flatten_maps=False)
```



[9]:

	Abc season Vmap(126)	Abc metadata Vmap(198)	123 match_week Int	Abc match_status Varchar(20)	123 match_id Int
1	{ "season_id": "38", "season_name": "20	{ "data_version": "1.1.0", "shot_fidelity_ve	6	available	68317
2	{ "season_id": "38", "season_name": "20	{ "data_version": "1.1.0", "shot_fidelity_ve	8	available	68318
3	{ "season_id": "38", "season_name": "20	{ "data_version": "1.1.0", "shot_fidelity_ve	10	available	68339
4	{ "season_id": "38", "season_name": "20	{ "data_version": "1.1.0", "shot_fidelity_ve	11	available	69164
5	{ "season_id": "38", "season_name": "20	{ "data_version": "1.1.0", "shot_fidelity_ve	12	available	68319
6	{ "season_id": "38", "season_name": "20	{ "data_version": "1.1.0", "shot_fidelity_ve	13	available	68354
7	{ "season_id": "38", "season_name": "20	{ "data_version": "1.1.0", "shot_fidelity_ve	14	available	68320
8	{ "season_id": "38", "season_name": "20	{ "data_version": "1.1.0", "shot_fidelity_ve	15	available	68321
9	{ "season_id": "38", "season_name": "20	{ "data_version": "1.1.0", "shot_fidelity_ve	16	available	68322

Rows: 1-17 | **Columns:** 14

Note that the number of columns have shrunk because the nested columns have coalesced into parent columns. Also note that the type of data type of a few columns have changed to Vmap.

VMap: Flexible maps that store the data as a string of maps, allowing the ingestion of data in varying shapes. The shape is not fixed and new keys can easily be handled. This is a great option when we don't know the structure in advance, or if the structure changes over time.

Another data type of complex data in VerticaPy is **Row**.

Row: More rigid because the dictionaries, including all the data types, are fixed when they are defined. Newly parsed keys are ignored. But because of it's rigid structure, it is much more performant than VMaps. They are best used when the file structure is known in advance.

In order to get data stored as rows, we have to switch True the **use_complex_dt** parameter. But currently, this is only allowed for data that is placed inside the Vertica server.

Other ingestion tips

Batch Ingestion:

Above are just a few basic ingestion involving only one file at a time. Sometimes you want to ingest a whol folder of files or even zipped files. Let's how we can do that using the ease of VerticaPy syntax.

Firstly, VerticaPy allows us to import a **batch of files** inside a folder of a particular format just by using the following syntax.

```
all_data=vp.read_csv("folder/*.csv")
```

OR below for JSON.

```
all_data=vp.read_json("folder/*.csv")
```

But in order to accurately extract all the data, the user has to make sure that all the files share the same structure i.e., they have the same column names and data types etc.

Acceptale File Formats:

Apart from the most popular formats JSON and CSV, VerticaPy allows ingestion of a lot of formats including:

1. AVRO
2. PARQUET
3. ORC
4. SHP

In the scenario where a user forgets the syntax of the particualr function, then VerticaPy has a generic function read_file function which automatically detecs the type of file and imports it.

Data augmentation/insertion:

In some instances, we already have a table and we want to add to that table. For such scenario, we can easily add more lines of data to an already existing file by using the **insert** parameter in any of the read functions introduced above. For example, for CSV we can do this:

```
data=vp.read_csv("gapminder.csv",
                table_name="master_data")
data.count()
```

	count
"country"	1704.0
"year"	1704.0
"pop"	1704.0
"continent"	1704.0
"lifeExp"	1704.0
"gdpPercap"	1704.0

Rows: 1-6 | **Columns:** 2

We see that we only have 1704 datapoints for each column. Now using the table_name parameter as a reference, we an insert more data into the same file using the following syntax:

```
data=vp.read_csv("gapminder.csv",
                table_name="master_data",
                insert=True)
data.count()
```

	count
"country"	3408.0
"year"	3408.0
"pop"	3408.0
"continent"	3408.0
"lifeExp"	3408.0
"gdpPercap"	3408.0

Rows: 1-6 | **Columns:** 2

Author Name: Umar Farooq Ghumman

Author Contact: umarfarooq.ghumman@vertica.com

Resources

- Gapminder Data
- Laliga Data