Data processing pipelines for Small Big Data

Esteban J. G. Gabancho - Anthony Franklin

$ whoami

- I write Python code for living (among other things)
  - Full-stack / DevOps
  - Data engineer
- Senior software engineer @ Fanalytical
- egabancho everywhere!


MY DATA
IS BIGGER THAN YOURS

What is big data?

Big data are high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.

Beyer & Laney, 2012

But seriously, what is big data?

- Nor primarily about data size
- **V**olume: The amount of data
- **V**elocity: The speed of data in and out
- **V**ariety: The range of data types and sources

Chances are your data is not that big at all

Problems with big data solutions

- Big data and big data solutions are expensive
- Specialized profiles
- Commercial solutions tend tide you in

You are better off without Big Data

The Small Big Data Manifesto

1. We (humans) produce more and more data every day.
2. Unless you work for Google, chances are your “big data” is not that big at all.
3. What used to be “big” yesterday is “large-ish” today and will be “small” tomorrow.
4. Definitions of “big data” usually refer to more attributes of the data than just sheer volume.
5. Big data technologies are great for data that are truly big.
6. Setting up a cluster of machines for many “big data” applications would be overkill and not financially viable.
7. Most users are stuck with laptops, workstations or individual servers.
8. The tools we have for those tend to break even for modest amounts of data.
9. People often use “big data” technologies on single machines, which is not efficient.
10. Ergo, we need new tools, inspired by the “big data” hype, that can process larger amounts of data without requiring the hardware- and management overhead of current “big data” technologies.
11. Many users of such tools would also lack experience of setting and running a data-intensive project.
12. Ergo, we need project management tools for such endeavours.

Small Big Data

- Usually fits on my computer's memory but expensive on the cloud
- Most of the time it'll be slower to use Big Data solutions

https://smallbigdata.github.io/index.html

`requirements.txt`

- Familiar with python should be enough to follow this presentation.
- Python 3.6+ (3.8.1)
  - Homebrew, pyenv, chocolatey, etc.
  - https://realpython.com/installing-python
- Not going talk about editors 🤗
- All the code examples can be found at github.com/egabancho/pydata-global-2020

Objectives

- Enhance memory consumption and speed
- Readability is much important that speed long term
- Avoid anti-patterns
- Before making something more efficient, make it work
- Won't cover garbage collection, C memory allocation, etc. into detail
- When in doubt try running `import this`

Python Freebies

Duck typing and Ask For Forgiveness

- Most times it's slower to check than to assume
- Most cases `try ... except ...` is faster than `if ... else …`
- Beware that handling exceptions is expensive
- most cases = most common case

Duck typing and Ask For Forgiveness

```python
class A:
    a = 'a'

class B:
    b = 'b'

def do_something_with_a_if_a(obj):
    if hasattr(obj, 'a'):
        return obj.a
    return None

def do_something_with_a(obj):
    try:
        return obj.a
    except AttributeError:
        return None
```

```
❯ python builtins/app_forgiveness.py
Done mostly As asking, in 0.1554 sec
Done mostly As w/o asking, in 0.1126 sec

Done mostly Bs asking, in 0.1438 sec
Done mostly Bs w/o asking, in 0.4784 sec
```

Iterators, generators, list comprehensions, and generator expressions

Maintain in memory what really is necessary at given moment in time

```python
# etl.py

import random
from typing import Generator


def extract() -> Generator[str, None, None]:
    """Dummy example return random integers between -1000 and 1000."""
    for i in range(1_000_000):
        yield random.randint(-1000, 1000)


def transform(value: int) -> str:
    """Fizz buzz."""
    if value % 3 == 0 and value % 5 == 0:
        return 'fizzbuzz'
    elif value % 3 == 0:
        return 'fizz'
    elif value % 5 == 0:
        return 'buzz'
    return str(value)
```

Iterators, generators, list comprehensions, and generator expressions

```python
data = list(extract())
process_data = [transform(d) for d in data]
```

```python
process_data = [transform(d) for d in extract()]
```

```python
process_data = (transform(d) for d in extract())
next(process_data)
```

```
❯ python builtins/app_process_list.py
Done, memory usage: 75.15 MB, in 1.29 sec

❯ python builtins/app_process_generator.py
Done, memory usage: 40.78 MB, in 1.23 sec

❯ python builtins/app_process_generator_expression.py
Done, memory usage: 0.01 MB, in 0.00 sec
```

Multiprocessing, Multithreading and AsyncIO

- Python is single threaded, which might lead to under-use resources
- Parallelism can lead to horrible headaches
  - GIL
  - Race conditions
  - Dead locks
  - …
- Used with “caution” can lead to big improvements

Multiprocessing, Multithreading and AsyncIO

```python
async def transform_chunk(values):
    await asyncio.sleep(len(values) * 0.0001)
    return [transform(value) for value in values]

def slow_transform(value):
    time.sleep(0.0001)
    return transform(value)
```

```
❯ python builtins/app_asyncio.py
Done single thread, in 136.99 sec
Done async, in 1.95 sec
```

The Usual Suspects

The usual suspects

- Common libraries present on most ETL/ML project: Numpy and Pandas
- Implemented in C
- Vectorized operations

- Others can help write better more efficient ETL pipelines

SQLAlchemy

- Object Relational Mapper (ORM)
- Pros
  - Python is not SQL
  - DB agnostic
  - Many queries will perform better
  - SQL injection
  - Advance features: connection pool, migrations, polymorphism, etc.
- Cons
  - Speed
  - Learning curve
  - Initial configuration overhead
  - Some queries might perform worse

SQLAlchemy

```python
class Customer(Base):
    __tablename__ = "customer"
    id = Column(Integer, primary_key=True)
    name = Column(String(255))

SIZE = 1_000_000

def test_sqlalchemy_orm_bulk_insert():
    _ = init_sqlalchemy()
    start_time = datetime.datetime.now()
    for chunk in range(0, SIZE, 10_000):
        DBSession.bulk_insert_mappings(
            Customer,
            [
                dict(name="NAME " + str(i), id=i + 1)
                for i in range(chunk, min(chunk + 10000, SIZE))
            ],
        )
    DBSession.commit()

    delta_time = datetime.datetime.now() - start_time
    print(
        'Done SQLAlchemy bulk insert,'
        f' in {delta_time.total_seconds():.2f} sec'
    )
```

```
❯ python libraries/app_sqlalchemy.py
Done plain sqlite, in 1.72 sec
Done SQLAlchemy ORM, in 42.25 sec
Done SQLAlchemy bulk insert, in 5.99 sec
```

Numba

- Just-In-Time (JIT) compiler
- Pros
  - Ease of use (`@jit`)
  - Automatic parallelization
  - Support for numpy
  - GPU support
- Cons
  - Many layers of abstraction make it very hard to debug and optimize
  - There is no way to interact with Python and its modules in `nopython` mode
  - Limited support for classes

Numba

```python
def distance(src, dst):
    result = np.zeros_like(src)
    for i in range(len(src)):
        result[i] = math.sqrt(
            (dst[i][0] - src[i][0]) ** 2 + (dst[i][1] - src[i][1]) ** 2
        )
    return result
```

```
❯ python libraries/app_numba.py
Done without numba in 2.53 sec
Done with numba  in 0.30 sec
```

Dask

- General purpose parallel programming solution
- Scales familiar analytics tools like Numpy, Pandas, and Scikit-Learn
- We love Dask!

```
import pandas as pd

df = pandas.read_csv('path/to/file.csv')
median_distance = df.groupby(df.account_id).distance.mean()


import dask.dataframe as dd
df = pandas.read_csv('path/to/file.csv')
median_distance = df.groupby(df.account_id).distance.mean().compute()
```

Dask

Compute duplicates and merge them, our success story

```
account_files = extract_accounts() # Extract accounts from source an save it to files

account_interim_files = (
    db.read_text(account_files)
    .map(json.loads)
    .flatten()
    .map(lambda record: translate_account(record))
    .map(json.dumps)
    .to_textfiles(f'data/interim/accounts-{now}-*.json')
)

new_accounts = (
    db.read_text(account_interim_files)
    .map(json.loads)
)
account_process_files = 'data/processed/accounts-*.json'
existing_accounts = (
    db.read_text(account_process_files)
    .map(json.loads)
)
all_accounts = db.concat([new_accounts, existing_accounts])

matcher = (
    all_accounts
    .map(extract_matching_info) # This function extracts account information used for matching, i.e. email
    .flatten()
    .to_dataframe(
        meta={'match_info': 'str', 'account': 'object'}
    )
)
all_accounts = matcher.groupby('match_info').apply(merge_customer)
all_acounts.map(json.dumps)
    .to_textfiles('data/processed/accounts-*.json')
```

Dask

Compute duplicates and merge them, our success story

- Less memory needed, 32Gb VM to 8Gb
- Less time 1 day -> 3 hours -> 30 minutes
- Smaller AWS bill 🎉

Feature Store

What is a feature store and why do I need it?

- It is like a data warehouse of features for machine learning
- 80% of data science is data wrangling -> precious work
- Many fields (sometimes expensive to calculate) are used in several models
- Better collaboration between team members
- Identify easier data drift (versioning)
- Michaelangelo (Uber), Hopsworks, Feast ...

Our opinionated implementation

- Based on already existing/inexpensive tools
  - Parquet files
  - Dask
  - AWS Athena
- Mostly for offline usage
- Stay tuned for a public release

Our opinionated implementation

Consumer high level API

```
fs = FeatureStore()
group = fs.get_group('customer')
group.calculate()
```

```
fs = FeatureStore()
group = fs.get_group('customer')
f_set = group.get_features(['distance', 'gender'], exclude=['*'], filters=['age >= 30'])
# f_set is a dask dataframe
```

Our opinionated implementation

Behind the scene

```python
def customer_general_features():
    stm = select(
    [...]

    ).where(...)
    df = get_data(stm)
    df = df.assign(
        distance=df[['lat', 'long']].apply(distance, meta=float)
        ...
    )
    ...
    return df
```

```python
FEATURE_STORE=dict(
    customer=dict(
        fields={
            'distance': float
            ...
        },
        functions=[
            'import.path.to.customer_general_info',
            ...
        ]
    )
)
```

Our opinionated implementation

Behind the scene

```python
for func in funcs:
    if isinstance(func, str):
        func = _import_string(func)
    tasks.append(unsync(func)())

it = iter(tasks)
ddf = next(it).result()
for task in it:
    ddf = ddf.merge(
        task.result(), how='outer', left_index=True, right_index=True
    )
```

```python
def create_table(
    db: str, name: str, path: str, columns_types: Dict[str, str], **kwargs: Any
) -> None:
    """Create a new Athena/Glue table."""
    if name not in [
        table["Name"] for table in wr.catalog.get_tables(database=db)
    ]:
        if 'id' not in columns_types:
            columns_types['id'] = 'string'

        wr.catalog.create_parquet_table(
            database=db,
            table=name,
            path=path,
            columns_types=columns_types,
            **kwargs
        )
```

Our opinionated implementation

Lessons learned

- Much more efficient now as a team, and in terms deployment resources
- Start small
- AWS Athena doesn't like dates
- When you say "we have a feature store" sounds very sexy ;-)

All Running Together

All running together

Airflow

- IMHO we were using Airflow wrongly for small big data ELTs
- Most of us use celery as to run the tasks, but that can lead to one worker one tasks type scenario
- We run Airflow on ECS and use celery but:
  - Each task uses the ECS operator to start a new container and run on "dedicated" resources
  - Celery runs on a smaller machine with hight concurrency
  - Bigger concurrency has shown smaller costs
  - Each tasks uses specific machine size depending on needs
  - No need to have client specific code deployed on airflow

All running together

How to achieve Airflow Zen: Docker

- Building python images
  - Remember each task will pull one of this and run something inside.
- Small fast builds mean faster deployments
  - Multistage builds to the rescue
- Sensible entry points
  - This is how you call your code, use something like click or typer

Conclusions

- Small big data is better than big data
- Store your precious features somewhere to reuse them
- Many tools and libraries can help you be more efficient
- Docker is key