Pandas and PostgreSQL

A comparison of dataframes and data stores

Lauren Gulland

Data store

repository for persistently storing and managing collections of data

Data store: PostgreS

repository for persistently storing and managing collections of data

- Customizable open-source RDBMS that extends the SQL language
- Only accepts writes on a single node → CP system
- Row-based storage

Data store: SQLite

repository for persistently storing and managing collections of data

- Small, embeddable RDBMS
- Stores all data in a single transferable file → CA system
- Memory-sqlite: in-memory system, like Pandas

Dataframe

2-dimensional container for labeled data

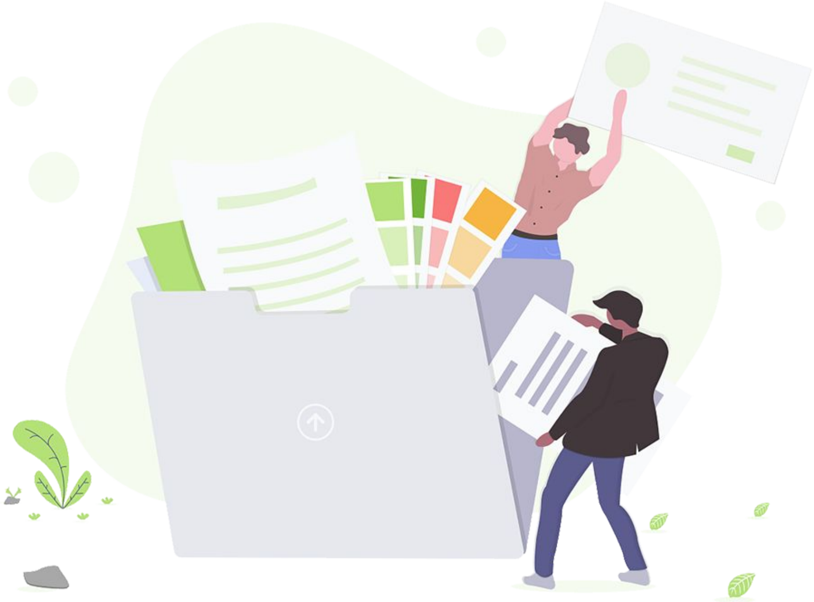
Dataframe: Pandas

2-dimensional container for labeled data

- Easy-to-use, fast data wrangling library in Python
- Initialized and stored in memory → CA System
- Columnar store

Comparison 1: Storage

How is data stored?



PostgreS: Partitioned Storage

- All tables stored as separate files in a PGDATA directory, indices stored as B-trees
- When a table exceeds 1 GB, it is divided into 1GB segments, to avoid file size limitations
- Rows are stored in pages in heap files

Pandas: Block Storage

	UID	Product	Quantity	Member	Promo	Points	Paid
Timestamp							
2015-07-01 02:50:00	xgy7b	A	3	True	SEA15	30	21.50
2015-07-01 03:30:00	sot5y	C	1	False	NaN	10	15.00
2015-07-02 03:52:00	g8z8l	B	2	False	NaN	30	10.25
2015-07-03 04:01:00	lxzuo	A	2	False	SEA15	15	16.25
2015-07-03 05:30:00	3peyj	C	4	True	BOB10	10	28.50

Axes

Timestamp
2015-07-01 02:50:00
2015-07-01 03:30:00
2015-07-02 03:52:00
2015-07-03 04:01:00
2015-07-03 05:30:00

UID
Product
Quantity
Member
Promo
Points
Paid

Blocks

	0	1	2
0	xgy7b	A	SEA15
1	sot5y	C	NaN
2	g8z8l	B	NaN
3	lxzuo	A	SEA15
4	3peyj	C	BOB10

	0	1
0	3	30
1	1	10
2	2	30
3	2	15
4	4	10

	0
0	True
1	False
2	False
3	False
4	True

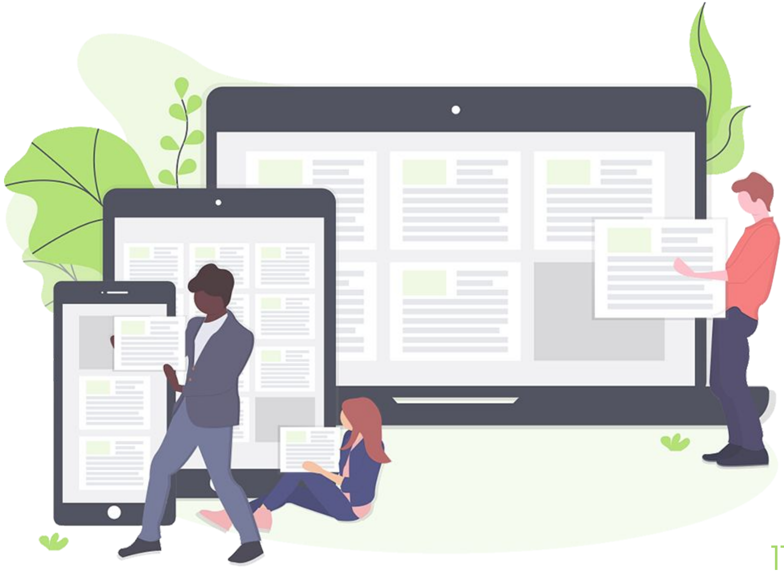
	0
0	21.50
1	15.00
2	10.25
3	16.25
4	28.50

Efficient at what?

- Operations within dtypes are quite fast
- Appending to the dataframe means copying every block

Comparison 2: Query Syntax

How do we access and manipulate the data?



Query Overview

SQL: Declarative

- Pseudo-English
- Compose the query with what you want (but not how to get it)

Python: Imperative

- Apply operations to the dataset
- Manually chain operations in the logical order they need to appear

Basic Select

SQL	Pandas

`SELECT id`
`FROM airports`
`WHERE ident="KLAX"`
	`airports[`
	`    airports.ident=="KLAX"`
	`].id`

Select with 2 Conditions

SQL	Pandas
SELECT * FROM airports WHERE iso_region = 'USA-CA' and type = 'seaplane_base'	airports[(airports.iso_region == 'USA-CA') & (airports.type == 'seaplane_base')]

GroupBy

SQL	Pandas

SELECT iso_county, type,
    count(*)
FROM airports
GROUPBY iso_country, type
ORDERBY iso_country, type
	airports,groupby([
	    'iso_country','type'
	]).size()

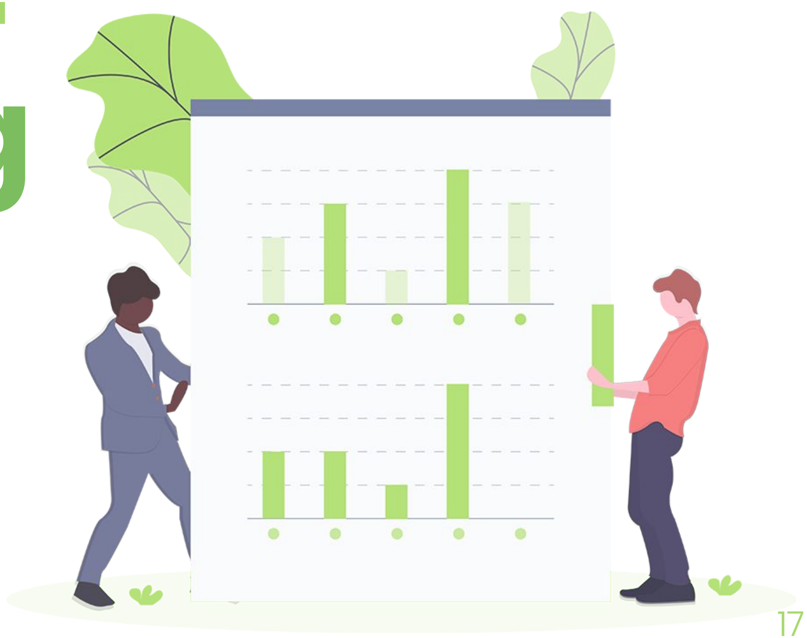
Inner Join

SQL	Pandas

SELECT airport_ident, type, description
 	
FROM airport_freq join airports on airport_freq.airport_ref=airports.id
 	
WHERE airports.ident='KLAX'
	airport_freq.merge(
	    airports[airports.ident=='KLAX'][['id]],
	    left_on='airport_ref',
	    right_on='id',
	    how='inner'
	)[['airport_ident','ident','description']]

Comparison 3: Benchmarking

What is each framework optimized for?



Test 1: Operations

	PostgreS			Pandas		
Operation	**Slowest**	**Fastest**	**Median**	**Slowest**	**Fastest**	**Median**
join	21.2	19.8	20.0	27.9	26.4	27.0
groupby	8.9	8.6	8.6	38.6	35.7	37.8
filter	10.2	9.5	9.7	27.5	25.0	25.3
sort	30.9	28.2	28.7	30.1	28.0	28.9

Test 2: Complex Queries

```
def do_it_in_sql():
    sql1 = """
    with recent_views as (
            select user_id, model_name
            from car_config_table
            where created_at > current_date - interval '2 months'
    ),
    popular_models as (
            select model_name as slug,
                   count(distinct user_id) configs
            from recent_configurations
            group by 1
            having count(distinct user_id) > 3
    ),
    popular_configurations as (
            select C.model_name, C.user_id, M.configs
            from recent_configurations C
            join popular_models M on M.slug = C.model_name
    )

    SELECT
    C1.model_name model_name,
    C1.configs model_configs,
    C2.model_name recommended_model_name,
    C2.configs recommended_model_configs,
    count(distinct C1.user_id) combo_configs
    from popular_configurations C1
        join popular_configurations C2 on C1.user_id = C2.user_id
    where
    C1.model_name <> C2.model_name
    group by 1,2,3,4
    order by model_name
    """
    df = execute_to_postgres(sql1)
    return df
```

```
def do_it_in_pandas():
    sql= """
    select user_id, model_name
    from car_config_table
    where created_at > current_date - interval '2 months'
    """
    df = query_postgres(sql)
    df['configs'] = df.groupby(['model_name'])
['user_id'].transform('nunique')
    df = df[df['configs'] > 3]
    crossdf = df.merge(df, on='user_id',how='outer')
    crossdf = crossdf[crossdf.model_name_y != crossdf.model_name_x]
    crossdf['combo_configs'] = crossdf.groupby(['model_name_x',
'model_name_y'])['user_id'].transform('nunique')
    crossdf = crossdf[['model_name_x', 'users_x', 'model_name_y',
'users_y','combo_configs']].drop_duplicates().sort_values('model_nam
e_x')
    return crossdf
```

```
sql2 = """
  with recent_configurations as (
    select user_id,
           model_name,
           COUNT(user_id) OVER (PARTITION BY model_name) as configs
    from car_config_table
    where created_at > current_date - interval '2 months'
    group by 1,2
  )

  SELECT C1.model_name model_name,
         C1.configs model_configs,
         C2.model_name recommended_model_name,
         C2.configs recommended_model_configs,
         count(distinct C1.user_id) combo_configs
  from recent_configurations C1
       join recent_configurations C2 on C1.user_id = C2.user_id
  where C1.model_name <> C2.model_name
  group by 1,2,3,4
  having count(distinct C1.user_id) > 3
 """
```

```
sql3 = """
 drop table IF EXISTS analytics.mv;
 create table analytics.mv as (
    select user_id,
           model_name,
           COUNT(user_id) OVER (PARTITION BY model_name) as configs
    from car_config_table
    where created_at > current_date - interval '2 months'
    group by 1,2);

 SELECT C1.model_name model_name,
        C1.configs model_configs,
        C2.model_name recommended_model_name,
        C2.configs recommended_model_configs,
        count(distinct C1.user_id) combo_configs
 from analytics.mv C1
 join analytics.mv C2 on C1.user_id = C2.user_id
 where C1.model_name <> C2.model_name
 group by 1,2,3,4
 having count(distinct C1.user_id) > 3
 """
```

Test 2: Complex Queries

	PostgreS			Pandas		
	Slowest	Fastest	Median	Slowest	Fastest	Median
Default	7.5	6.9	7.0	6.7	5.8	6.0
Optimized (V2)	6.6	6.1	6.2	-	-	-
Optimized (V3)	7.4	6.4	6.6	-	-	-

So….why?

- PostgreS is doing a lot more for us behind the scenes.
- SQL isn’t made for complex mathematical operations, even with User Defined Functions.
- Benefits of keeping more tasks on SQL can go far beyond execution time.

Conclusions

What’s next?

Further Directions:

- Custom benchmarking of queries with SQLite
- Comparisons between Pandas and In-memory SQLite?
- Limitations of the `df.to_sql()` function in Pandas, and more on Pandas-SQL interfaces