Querying data with SQL and Pandas

First, SQL...

- SQL has been king of the relational (tabular) database landscape since the 70s/80s
- Grandfather of relational architecture and standard data manipulation practices (joins, aggregations, groupings, pivoting/melting).
- People who come from a deeper SQL background will tell you that interpreted languages cant compete with SQL from a speed/performance perspective.
- BUT...
    - Comparing performance of sqlite and python-pandas | Tableau Public

SQL vs Python performance, continued

- NOT SO FAST… a newcomer joins the rumble:
  - Efficient SQL on Pandas with DuckDB - DuckDB
- BUT WAIT, THERE'S MORE:
  - Polars

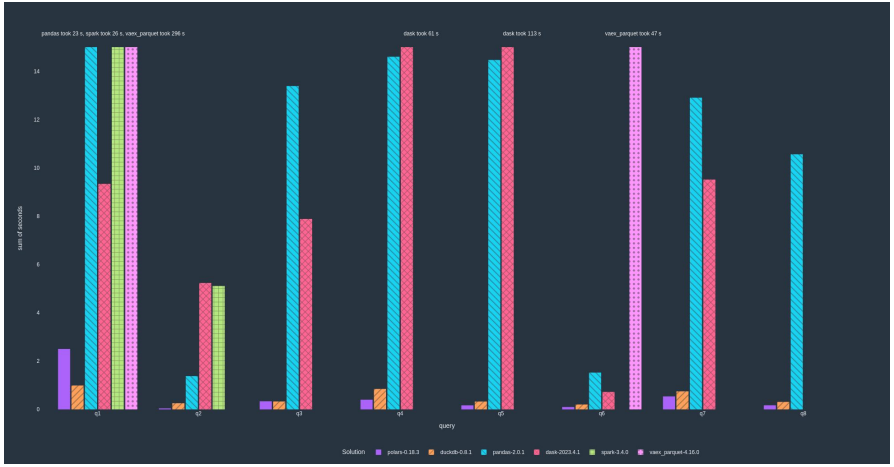
By the way…. What is a data frame, exactly?

Aggregation

Name	Time (s)
DuckDB (1 Thread)	0.60
DuckDB (2 Threads)	0.42
Pandas	3.57
Pandas (manual pushdown)	2.23

Joins

Name	Time (s)
DuckDB (1 Thread)	1.04
DuckDB (2 Threads)	0.89
Pandas	20.4
Pandas (manual pushdown)	3.95



Other considerations: SQL vs Python

SQL

- Pros:
  - Stability and reliability – Truly battleground tested, ubiquitous
  - Made for querying
  - Offload compute to dedicated servers
  - Secret sauce- the query optimizer
- Cons:
  - Complexity creep
  - Limitations of optimizer
  - Modern software/devops tools in SQL world have been slow to catch up to “AppDev” world.
  - Learning curve

Python

- Pros:
  - Syntax, developer tooling/friendliness
  - Broader scope (web, ML, plotting, etc)
  - Rate of innovation
- Cons:
  - Complex queries may need manual tuning
  - Not as popular in data work

No matter what language you work in, careful architecture is more important than raw computing power in most real-world applications.

Syntax: SQL vs Pandas

```sql
SELECT
DISTINCT
    SUM(cont.Amount) AS Total
FROM mec.Contributions AS cont
    LEFT JOIN mec.Committees AS cmt
ON cont.MECID = cmt.MECID
WHERE cmt.Name = "Cori Bush for Congress"
GROUP BY cont.City
HAVING Total > 1000
ORDER BY Total
```

```python
import pandas as pd

contributions = pd.DataFrame()
committees = pd.DataFrame()

bush_cmte = committees[committees["Name"] == "Cori Bush for Congress"]

combined = contributions.join(committees, on='MECID', how='left')

totals = combined.groupby("City")["Amount"].sum().rename(columns={"sum": "Total"})

totals = totals[totals["Total"] > 1000].sort_values("Total", ascending=False)
```

"Order of execution"

1. FROM clause
2. ON clause
3. OUTER clause
4. WHERE clause
5. GROUP BY clause
6. HAVING clause
7. SELECT clause
8. DISTINCT clause
9. ORDER BY clause
10. TOP clause

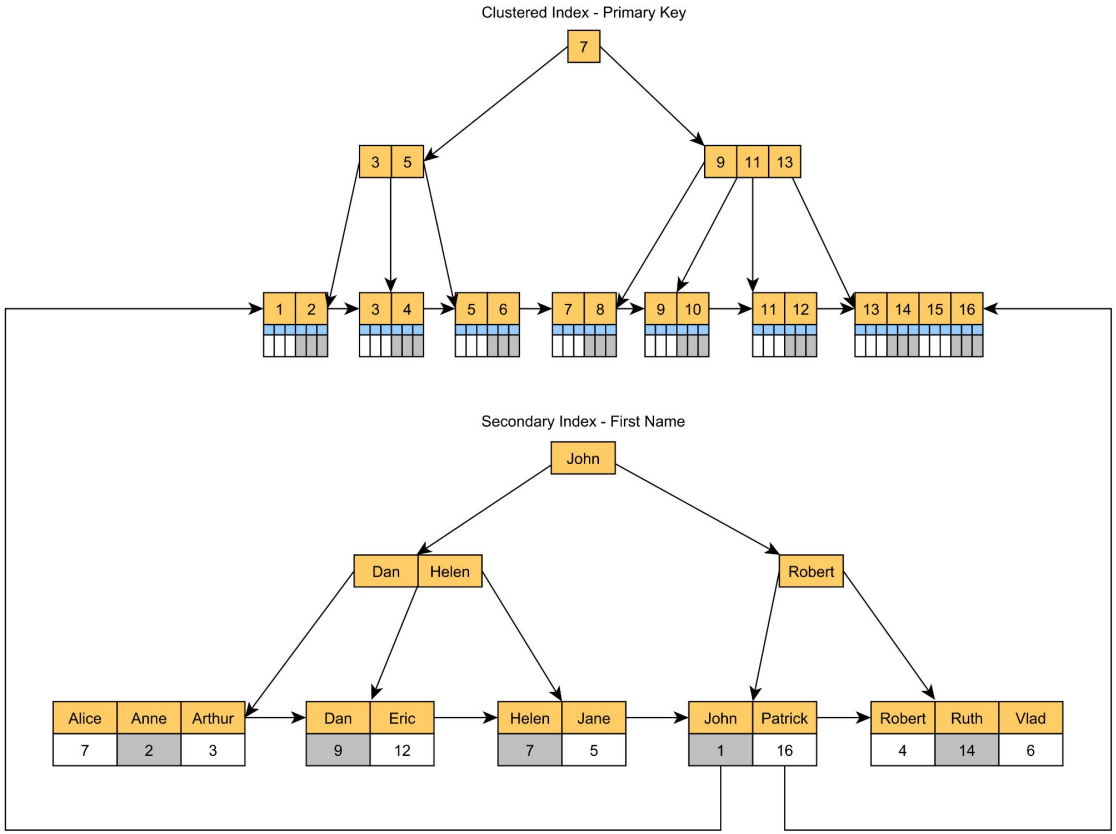
```python
import pandas as pd

pd.DataFrame(
    {"A": [1, 2, 3], "B": [4, 5, 6]},
    index=pd.Index(["First", "Second", "Third"], name="position"),
)
```

✓ 0.8s

position	A	B
First	1	4
Second	2	5
Third	3	6

Indexing & Constraints

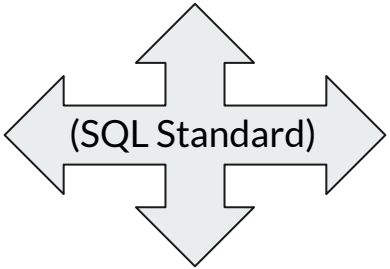
Source: [The best UUID type for a database Primary Key | Vlad Mihalcea]

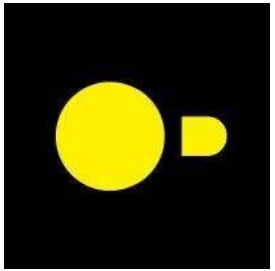


It all comes down to CRUD and ACID

- CRUD operations
  - Create
  - Read
  - Update
  - Delete
- ACID transactions
  - Atomicity - All or nothing
  - Consistency - Data is correct
  - Isolation - Independent of other transactions
  - Durability - changes are persistent

RDBMS systems



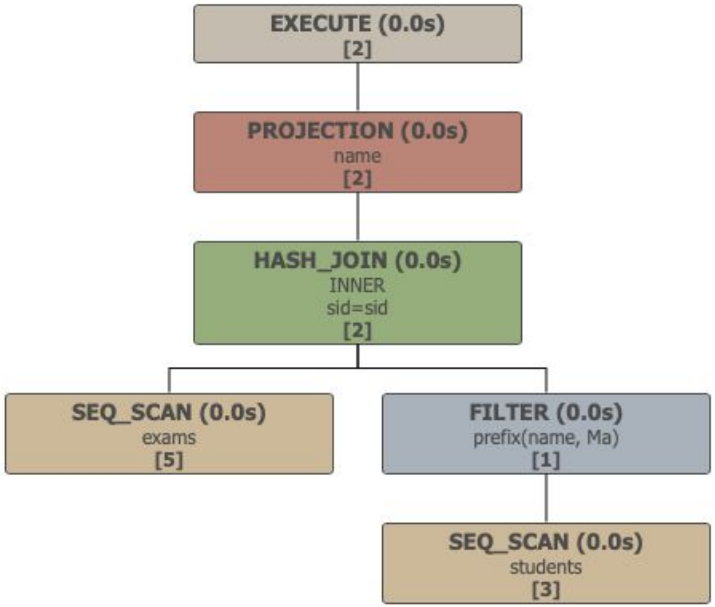


Differences:

- User Permissions/Security
- Programmatic features
- Some architecture decisions
- Optimizer Secret Sauce

Under the hood - The query optimizer

DuckDB



MS SQL Server