Jan 10, 2026

Tags: Python, SQL, revenue, budget, costs, Spark SQL, PostgreSQL, SQL Server, dbt, data build tools, CapEx, OpEx, EC2, AWS,

Python & SQL – Usage, Costs, the Good & Wasted

Beginning 2

Overall up front: 2

- **Python vs. SQL – Cost Advantages** 3
- **Python vs. SQL – Cost Wastes** 6

A Decent Cost Comparison Table…? 8

- **Check this out on SQL – CTEs / Temp Tables / CTAS** 9
  - **Temp View** 9
  - **Temp Tables — Physical Snapshots** 9

Some Rules of Thumb: 11

Python Dataframe merge vs **SQL Joins** 12

Summary 13

Definitions 13

Beginning

Sooooo. Using Python and SQL and the costs are adding up eh…? Basically, we need to know about cost models for each area.
Well, for the inexperienced, here is a 'bit' of content to keep in mind to avoid racking up major unnecessary costs.

Much of the material in this paper is basically research I've done to see where there might be better options in different situations. The research is primarily from multiple articles out there but I will also check out a few of the AI chatters to see what they feed back to me.

Over time, I will add more research / experience content here for other new Data Engineers (or ML Engineers).
But, if'n youse seez sumthin' that is wrong or incorrect – feel free to lemme know and explain why youse believe it to be so…
(*One must have humor from time to time; it makes work transpire much easier*… and I have been in MUCH more serious work situations while a Marine a long time ago and more recently while I was working as a civilian/fed intelligence officer.)

Overall up front:

- **SQL** is less expensive for moving, filtering, and aggregating data
- **Python** is less expensive for orchestration, complex logic, and non-relational work
- The ***biggest waste*** is using Python to do what SQL engines **already does less expensively**

Python vs. SQL – Cost Advantages

Aspect	Python	SQL

Costs, Costs, Costs….	**Complex logic/transformations** on **small-to-medium data**
	• **No query costs** if data is already local
	• **Reusable code without re-querying**
	• **Free compute on your local machine**
	Iterative development/prototyping
	• **No per-query costs** like cloud DBs
	• Can **manipulate data in-memory repeatedly**
	Where Python Has the BIGGEST Cost Advantage
	Orchestration & glue logic
	Python is cost effective for:
	• Control flow
	• Conditional logic
	• Pipelines
	• API calls
	• Multi-system integration
	Cost:
	For large datasets:
	• **Python** = **dollars to hundreds of dollars**
		Working w/ large datasets in DBs (*database*)
		• **SQL runs on the DB server** (*close to data*) – i.e.: SQL engines (Redshift, Athena, BigQuery, Snowflake)
		• **No data transfer costs** - processing happens where *that* data lives – less network i/o
		• Highly optimized for filtering / aggregating millions of rows – i.e.:
		  ○ Filtering 1 billion rows in Athena costs pennies vs loading into Python
		Cost:
		• Scans only required columns
		• Executes in compiled engine
		• Parallelized automatically
		Less expensive than pulling data into Python.
		For large datasets:
		• **SQL = pennies**
		(*Still, it will depend on task…*)
		Serverless SQL is brutally cheap, i.e.:
		• Athena
		• BigQuery on-demand
		Cloud data warehouse queries (*like in Athena usage*)
		• Only pay for data scanned, i.e.: using:
		• **TABLESAMPLE SYSTEM** (10) in SQL = **90% cost savings immediately**
		Efficient WHERE clauses reduce scanned data
		Or using specific column selection:
		Example:
		`SELECT class, AVG(sepal_length)`
		`FROM iris`
		`GROUP BY class;`

Aspect	Python	SQL
Costs, Costs, Costs (cont.) …	**Small to medium data transformations** If data: • Fits in memory • Is already filtered • Needs complex logic **Python** (pandas, numpy) is cheaper than spinning up: • Spark Warehouses • Distributed SQL **Non-tabular data** Python wins for: • JSON XML Images • NetCDF Logs ML features	Cost is usually: • per TB scanned • No idle compute cost If our query scans 5 GB: You pay for 5 GB — not for a running server.
Flexibility	It is very versatile and handles various data types & formats; allowing for broader implementation in data science, automation & web applications. Scales linearly with data size Python: • Interpreted • GIL-bound • Memory-heavy	It is specialized for relational DB operations, making it efficient for data manipulation & retrieval. Scales linearly with data size SQL engines: • Are written in C/C++ • Vectorized • Highly optimized

Aspect	Python	SQL
Flexibility	It is very versatile and handles various data types & formats; allowing for broader implementation in data science, automation & web applications. **Scales linearly** with **data size** Python: • Interpreted • GIL-bound • Memory-heavy	It is specialized for relational DB operations, making it efficient for data manipulation & retrieval. **Scales linearly** with **data size** SQL engines: • Are written in C/C++ • Vectorized • Highly optimized
Development Costs	It often requires fewer lines of code for complex tasks due to its readability and powerful libraries (e.g., Pandas, NumPy), which can reduce development time.	It can be more straightforward for querying DBs, which can lead to faster execution of specific data retrieval tasks without extensive coding.
Community & Libraries	A large array of libraries & frameworks is available for numerous tasks, reducing the need for custom development.	SQL's standardized queries are supported across various DB systems, lowering training costs for teams familiar w/ SQL syntax.
Learning Curve	Although Python has a learning curve, it is generally considered easier for beginners, which reduces training costs.	SQL is straightforward for data querying, but advanced features and optimizations may require deeper expertise, potentially increasing training expenses.

Python vs. SQL – Cost Wastes

Python

- **Performance Overhead**: Python may experience slower performance for large datasets compared to SQL. And if you do not optimize it, it will lead to higher computational costs.
- **Library Management**: Over-reliance on multiple libraries can lead to version conflicts and increased maintenance costs. ***A major headache for everyone…***
- **Inefficient Code**: Poorly written Python code leads to longer execution times, demanding more computational resources and increasing operational (OpEx) costs. **Possibly,** OpEx budget might balloon but 'could' still be better than CapEx costs…

Loading massive datasets into memory

```
# EXPENSIVE: Pulls 10 GB into memory/network
df = pd.read_sql("SELECT * FROM big_ole
_table", connection)

# BETTER: Filter in SQL first
df = pd.read_sql("SELECT * FROM big_ole
_table WHERE ...", connection)
```

Data transfer costs

- Moving data from cloud → local machine **= bandwidth costs**
- Processing 100 GB locally vs in-DB = huge transfer fees

Inefficient loops on large data

```
# Slow & Expensive (CPU time)
for row in df.iterrows():   # Don't do this!
    process(row)

# FAST & Less expensive
df['result'] =
df.apply(vectorized_function) # Vectorized
operations
```

SQL

- Licensing Costs: Some SQL DBMS (*database management systems*) may involve licensing fees, leading to higher OpEx costs compared to open-source alternatives.
- Complexity in Scaling: As DBs grow, scaling SQL solutions can become expensive and complex, especially if additional features or optimizations are needed.
- Training and Expertise: Advanced SQL queries and optimizations may require specialized knowledge, leading to higher labor costs for skilled personnel.
  - Here, this could add to OpEx budget more than expected…

One that everyone should already know about – the use of the infamous '*****'

Full table scans without filters

```
-- Expensive: Scans entire table
SELECT * FROM big_ole_table;

-- Less expensive: Only scans needed data
SELECT * FROM big_ole_table WHERE date =
'2026-01-10';
```

Not using partitioning in cloud databases

- Athena charges by **data scanned**
- **Querying unpartitioned tables = scanning everything**

Repeated identical queries

- **Each query costs money** in **cloud DBs**
- **Should cache results** or **use materialized views**

SELECT * when you **only** need a few columns

```
-- Wastes money scanning unused columns
SELECT * FROM table;

-- Better
SELECT col1, col2 FROM table;
```

Python	SQL

Not using appropriate compute resources

- Running big jobs on expensive EC2 instances when Lambda / Glue would be cheaper
- Keeping instances running 24/7 for occasional processing ()

(This is an area at one time, where I was running point at an intel agency. To have all divisions use AWS cost explorer and Bill & Management – to review and revise their use of EC2s [*many had EC2s that were **complete overkill** for the task at hand, especially for DevOps & Testing*]**, DBs and S3 storage** [many had **100s** of **TBs** of data needlessly in Standard – not being accessed]**).**

Few months later, final review: **$100s** of **Ks / month saved**...

Biggest Cost WASTE with Python

Pulling large datasets out of DBs **-- Classic mistake**:

```
df = pd.read_sql("SELECT * FROM
big_ole_table", conn)
df = df.groupby("class").mean()
```

WHY is it expensive...?

- Full table scan — Network transfer
- Python memory usage — Longer runtimes
- Bigger instances

FIX:

```
SELECT class, AVG(sepal_length)
FROM huge_table
GROUP BY class;
```

Cost-effective workflow:

1) **SQL**: Filter/aggregate/sample in Athena (cheap, close to data)
2) Export: Only move reduced dataset
3) **Python**: Complex visualizations/analysis locally (free)

i.e.:

```
SELECT lat, lon, temp_c, forecast_step
FROM mogreps_analytics.mogreps_latest_6h
TABLESAMPLE SYSTEM (10)    # 90% cost
savings!
WHERE forecast_step < 6    # Further
reduces scanned data
```

SQL is **terrible** at:

1) Loops
2) Branching
3) External APIs
4) Complex business logic

Trying to do 1 - 4 in SQL leads to:

- Stored procedure abuse
- Long-running warehouse sessions
- Developer inefficiency (which is a real cost)

SQL engines either:

- Scan too much
- Require expensive UDFs
- Force schema contortions

Python	SQL

Using Python where vectorized SQL exists

(a) Examples: • Filtering • Aggregations & Joins • Deduplication	(b) Python does these: • Slower • More memory • More compute cost
(c) Overusing Spark/Glue for small jobs Running Glue for: • 10 MB .csv • Simple aggregation	(d) Cost waste: • Job spin-up • Executor overhead • Logging overhead **Lambda + SQL query would be cheaper**

Cost wasteful workflow:

1) Worst ➔ **SELECT ***
2) Pull **entire table** into Python or
   - pulling **raw data** instead of aggregated data
3) Filter/sample in pandas
4) Pay for data transfer AND compute

Do control logic in Python

Python minimizes developer and orchestration cost.

As some say, **"Let Python orchestrate — not compute."**

AND NEVER… Never move large data just to "feel comfortable."

Using SQL warehouses as application logic engines:

(a) Running: • Long sessions • Stored procedures w/ loops • Repeated transformations	(b) Results: • Idle compute • Expensive warehouse uptime • Locked resources

(c) Re-scanning massive tables repeatedly
`SELECT ...`
`FROM fact_table`
`WHERE date = '2024-01-01';`
Run 100 times:
• We pay for 100 scans
	(d) Fix
	• Materialized views
	• Incremental tables
	• Partition pruning

(a) **Doing ML or complex math in SQL – bad…** • Iterative algorithms • ML training • Feature engineering with state	(b) Result: • Long-running queries • High warehouse costs • Poor maintainability **Python is cheaper here.**

Do data reduction in SQL

SQL minimizes compute cost per byte.

A Decent Cost Comparison Table…?

Task	Cheaper Tool	Why
Filtering rows	SQL	Predicate pushdown
Aggregations	SQL	Vectorized execution
Joins	SQL	Optimized planners
API calls	Python	SQL can't
Complex business rules	Python	Readability + speed
ETL orchestration	Python	Control flow
ML training	Python	Libraries + GPUs
BI datasets	SQL	Columnar + caching
Small ad-hoc scripts	Python	No warehouse spin-up

Check this out on SQL – CTEs / Temp Tables / CTAS

CTEs (*common table expression*) – every single time that CTE is run (*in the overall use of one application*), we are:
NOTE: The CTE is a **query rewrite** AND a **logical alias**

a) creating something that is run every time it is called,
b) *s-l-o-w-i-n-g* down the application processing time (*resource usage – data shuffling & I/O costs*),
c) every time it is called in the code, we undergo computational costs, needlessly,
   - ESPECIALLY when they use a **SELECT *** statement…
   - But. The SQL CTE is not inherently 'always' creating more computation than a Temp View
d) The CTE is scoped to one query and disappears after execution
e) And as one person wrote in a comment (*Portuguys on Medium*) – "it's not a binary choice: never / always use CTEs – **the scenario is, it depends**."

```
WITH my_cte AS (
    SELECT * FROM large_table WHERE condition
    )
SELECT * FROM my_cte;
```

Engines that **materialize CTEs** by **default**	
Some systems (historically): • Older PostgreSQL • Some MPP engines • Certain warehouse configurations	They treat CTEs as **optimization fences** Result: • Forced materialization • Extra I/O

Temp View

- **Session-scoped** - persists across multiple queries
- **Not materialized** (if it's a VIEW) - also recomputed each time
- **Reusable** - can query it multiple times in different statements
- But – **still not stored *unless*** you use:
  - `CREATE TEMP TABLE AS`

```
CREATE TEMP VIEW my_view AS
SELECT * FROM large_table WHERE condition;

SELECT * FROM my_view;
```

Caching **Temp views/tables** avoids repeated scans/shuffles when performance is a priority.
The catch, thinking this way may be more applicable for **PostgreSQL** more than for **SQL Server**.
But. CTEs and Temp Views are about the same. Neither of the two are materialized (*not physically stored*), so any computation occurs when a query is called.
What CTEs have going for it is that they ***are*** very readable for clear thought reasoning (*easy explanation*)…

Temp Tables — Physical Snapshots

Sometimes, you need intermediate results to *persist physically* for debugging or reuse across sessions.
That's when **TEMP tables** shine.

Materialized vs Non-Materialized		
Type	**Materialized?**	**Computation**
CTE	No	Computed every reference
Temp VIEW	No	Computed every reference
Temp TABLE	**Yes**	Computed **once**, then **stored**

CTEs vs	**TEMP Tables/CTAS**
Use CTEs when: • referencing result **once** or **twice** • query is simple and fast • want clean, readable code	**Use Temp Tables/CTAS** when: • querying result **many times** • computation is **expensive** • working w/ **large datasets** • Incremental pipelines

Use CTEs when:	**Use Temp Tables/CTAS when:**
• We reference the result **once** or **twice** • The query is simple & fast • We want clean, readable code	• We query the result **many times** • The computation is **expensive** • working w/ **large datasets**

CTE Uses MORE Computation:	TEMP Tables – a better approach:
-- BAD: CTE referenced 3 times = computed 3 times WITH expensive_cte AS (SELECT complex_join(...) FROM massive_table) SELECT * FROM expensive_cte UNION ALL SELECT * FROM expensive_cte -- Recomputed! UNION ALL SELECT * FROM expensive_cte; -- Recomputed again!	-- GOOD: Materialized once CREATE TEMP TABLE expensive_result AS SELECT complex_join(...) FROM massive_table; SELECT * FROM expensive_result UNION ALL SELECT * FROM expensive_result -- Just reads stored data UNION ALL SELECT * FROM expensive_result;

Expensive Cost implication:	Better approach efficient in Athena:
-- If you reference a CTE multiple times: WITH sampled_data AS (SELECT * FROM mogreps_latest_6h #a View of mine TABLESAMPLE SYSTEM (10)) SELECT * FROM sampled_data WHERE condition1 UNION ALL SELECT * FROM sampled_data WHERE condition2; **This might scan the base table TWICE** (depending on query optimizer)	-- Create a real table (one-time scan cost) CREATE TABLE mogreps_sample AS ---- # a table of mine SELECT * FROM mogreps_latest_6h---- # a View of mine TABLESAMPLE SYSTEM (10); -- Now query the smaller table multiple times (cheap!) SELECT * FROM mogreps_sample WHERE condition1; SELECT * FROM mogreps_sample WHERE condition2;

In using AWS Athena	
Athena does NOT support	**Athena DOES support**
• Temp tables • Temp views • Materialized views	• CTEs (WITH clause) • Creating real tables with ○ CREATE TABLE AS SELECT

Cost comparison summary

Scenario	Cheaper	Why
Single-use logic	CTE	Inlined
Simple filtering	Either	Same plan
Reused many times	Temp table	One compute
Large expensive joins	Temp table	Avoid recompute
Glue / Spark	Same	Logical rewrite
Warehouses w/ materialized CTEs	Temp table	Predictable cost

Some Rules of Thumb:

- **"Push computation to the data"** (use SQL where data lives)
- **"Pull only what you need"** (filter/sample before extraction)
- **Use Python for logic SQL can't handle** (ML, complex transformations, visualizations)

For a Date Engineer project I did, I used AWS Lambda + Glue (ELT) + S3+ Athena. Here, I chose to do reduced sampling (10%) instead of using the entire dataset. In this case, a Glue Python Shell job was better than the Spark Job. The source dataset was only ~22 MB but it was in .nc format, which when expanded, balloons out into a **6.6 GB dataset** (.csv). So, 10% sampling in this scenario...

For visualization, after the ELT job, doing a 10% sampling in SQL was more cost-effective, for me, than pulling everything and sampling in Python.

Python Dataframe merge vs SQL Joins

```python
# Prepare data using Python Dataframe merge
streams['listen_date'] = pd.to_datetime(streams['listen_time']).dt.date
merged_data = streams.merge(songs, on='track_id', how='left')

# Compute each KPI
# KPI 1: Daily Genre Listen Count
genre_listen_count = merged_data.groupby(['listen_date',
'track_genre']).size().reset_index(name='listen_count')

# KPI 2: Average Listening Duration per Genre per Day
merged_data['duration_seconds'] = merged_data['duration_ms'] / 1000
avg_duration = merged_data.groupby(['listen_date',
'track_genre'])['duration_seconds'].mean().reset_index(name='average_duration')

# KPI 3: Daily Genre Popularity Index
total_listens = merged_data.groupby('listen_date').size().reset_index(name='total_listens')
genre_listen_count = genre_listen_count.merge(total_listens, on='listen_date')
genre_listen_count['popularity_index'] = genre_listen_count['listen_count'] /
genre_listen_count['total_listens']

# KPI 4: Most Popular Track per Genre per Day
most_popular_track = merged_data.groupby(['listen_date', 'track_genre',
'track_id']).size().reset_index(name='track_count')
most_popular_track = most_popular_track.sort_values(by=['listen_date', 'track_genre',
'track_count'], ascending=[True, True, False])
most_popular_track = most_popular_track.drop_duplicates(subset=['listen_date', 'track_genre'],
keep='first').rename(columns={'track_id': 'most_popular_track_id'})

# Combine all KPIs into one DataFrame
final_kpis = genre_listen_count[['listen_date', 'track_genre', 'listen_count',
'popularity_index']]
final_kpis = final_kpis.merge(avg_duration, on=['listen_date', 'track_genre'])
final_kpis = final_kpis.merge(most_popular_track[['listen_date', 'track_genre',
'most_popular_track_id']], on=['listen_date', 'track_genre'])

final_kpis.head(25)
```

multiple queries & merges

- Compute ea. metric separately (listen count, avg duration, most popular track)
- Then merge together
- Creates a clean, comprehensive KPI dashboard dataset

```python
# Prepare data using SQL equivalent code:
sql_query = """
SELECT
    glc.listen_date,
    glc.track_genre,
    glc.listen_count,
    glc.popularity_index,
    ad.avg_duration,
    mpt.most_popular_track_id
FROM genre_listen_count glc
LEFT JOIN avg_duration ad
    ON glc.listen_date = ad.listen_date
    AND glc.track_genre = ad.track_genre
LEFT JOIN most_popular_track mpt
    ON glc.listen_date = mpt.listen_date
    AND glc.track_genre = mpt.track_genre
 print(sql_query)
```

one big query

Summary

Python might offer more: flexibility, ease of use, and a broad range of applications. Whereas, SQL excels in structured data retrieval and manipulation.
Each tool comes with its very own cost advantages and potential waste, making the company's choice of which, very dependent on project-specific requirements and long-term resource management strategies.
For every task, careful consideration and expected scalability can help mitigate unnecessary CapEx AND OpEx costs in both Python and SQL environments.

Definitions

CapEx: capital expenses, traditional – one-time purchases:

- buying servers/hardware; building a data center; purchasing software licenses (perpetual); buying a company car – basically, large upfront costs, that depreciate over time with values also decreasing…

OpEx: operating expenses – to run business day-to-day:

- rental/leasing costs for infrastructure, equipment insurance, site lease/rent, cloud services (AWS, Azure), i.e.: Glue/Athena query costs, monthly software subscriptions (Office 365, Salesforce), personnel (salaries), utilities (heating/cooling/power), internet…
- regular recurring payments; fully deductible in the year spent; no ownership – we're renting/subscribing; scales with usage (pay as you go)

Making good use of **dbt** (*data build tools*) is a good SQL tool, though I, myself, have not had much use of it yet. I am going by a couple of folks who make the tool credible.
But I do know that we have to be careful with **dbt**, regarding the introduction of additional dependencies.

Temporary views from **Spark SQL** can be connected to caching to avoid repeated computation of heavy intermediate logic.