Unlearning SQL

Leverage the SQL design patterns in Python

S.Lott

https://fosstodon.org/@slott56

https://github.com/slott56

14-May-2025

Table of Contents

SQL is Helpful

Folks know SQL.

- They are fluent in SQL's design patterns.
- Some find it hard to convert SQL designs to Python.

This talk should help clarify SQL from a Python perspective.

Database Overuse

Given a processing problem...

Database Overuse

Given a processing problem...

1. Define (and normalize) tables
2. Write, and debug a load script
3. Write, and debug the SQL

Easy, right?

Database Overuse

Given a processing problem...

1. Define (and normalize) tables
2. Write, and debug a load script
3. Write, and debug the SQL

Easy, right?

Maybe not

SQL Overheads

The database engine has overheads

Lots of them.
Locking. Storage management. Permissions. Serialization.

War Story:
Developer struggling with transient data processing.
The app does repeated **Create-Load-Query-Drop** cycles.
The `DROP` (it turns out) is both unpredictable and slow.

(Even SQLite introduces overheads.)

How do we unlearn SQL?

Two steps to moving past SQL:

1. Understand the SQL design patterns.
2. Rework those design elements in Python.

Consider the core Select statement:

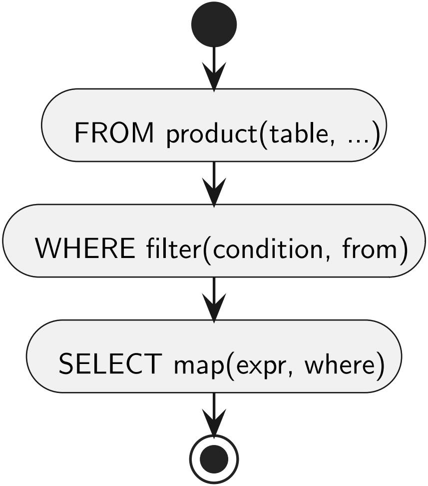
`SELECT` *expr, ...*

`FROM` *table, ...*

`WHERE` *condition*

We'll get to `GROUP BY` and `HAVING` later.

SELECT works like this



SELECT in Python


```
# FROM t1, t2, ...
from_ = itertools.product(t1, t2, ...)

# WHERE c
where = (row_tuple
    for row_tuple in from_
        if c(row_tuple))

# SELECT ex1, ex2, ...
result = list(
    (ex1(row_tuple), ex2(row_tuple), ...)
    for row_tuple in where)
```

Good and Bad

- The SQL maps directly to Python code.
- Python has more syntax:
  `WHERE` *expr* becomes `(r for r in from_ if` *expr*`)`.
- `SELECT expr, expr, expr` is even **more** complicated-looking.

Let's look at details.
All the syntax means there are a lot of places to add processing.

The From Clause

Each of the FROM tables needs to be iterable sequences.
`list[dict[str, Any]]`

```
from itertools import product
from_ = product(t1, t2, t3)
```

Yes. It's the Cartesian product.

Aha!

"Gotcha!"

- A *real* database doen't do cartesian products all the time.
- It has fancy query algorithms and optimizations.

Aha!

"Gotcha!"

- A *real* database doen't do cartesian products all the time.
- It has fancy query algorithms and optimizations.

"Your nonsense is clearly unworkable in general."

Query optimization

- Requires extended syntax to suggest query optimizations.
- Require someone to design the right indexes.
- Requires detailed statistics on key distribution.

You **can** do query optimization in Python, also.

Query optimization

- Requires extended syntax to suggest query optimizations.
- Require someone to design the right indexes.
- Requires detailed statistics on key distribution.

You **can** do query optimization in Python, also.

We'll get to it.

The Where Condition

Context:

```
where = (row for row in from_ if c(row))
```

Or.

```
where = filter(c, from_)
```

```
def c(row: tuple[dict[str, Any], ...]) -> bool:
    t1, t2, t3 = row
    return (
        t1['rowid'] == t2['foreign_key']
        and t2['some_key'] == t3['whatever']
    )
```

The Select Clause

Context:

```
result = list(
    (ex1(row_tuple), ex2(row_tuple), ...)
    for row_tuple in where)
```

Or.

```
result = map(row_builder, where)
```

```
def ex1(row: tuple[dict[str, Any], ...]) -> Any:
    t1, t2, t3 = row
    if t1['value'] % 2 == 0:
        return t1['value'] // 2
    else:
        return t1['value'] * 3 + 1
```

Key Point 1

You're not limited

You don't have to write SQL expressions.

You have the **Vast Python Ecosystem** available.

Key Point 1

You're not limited

You don't have to write SQL expressions.

You have the **Vast Python Ecosystem** available.

(Turn on echo effect)

Unlimited Computing Power!

Key Point 1

You're not limited

You don't have to write SQL expressions.

You have the **Vast Python Ecosystem** available.

(Turn on echo effect)

Unlimited Computing Power!

The `From` clause needs work.

The Cartesian Product Problem

Databases do “cart-prod” joins all the time.

Very small tables are easier to fetch from disk into cache ignoring any indexes.

Two common alternative algorithms:

- Sort-Merge Join
- Lookup Join

Going to take a shallow look at each.

Sort-Merge Join

For very large tables.

1. Sort each table into a consistent order by the join key for that table.
   May need multiple files and the os utility **sort** command.
2. Create row tuples for matching rows from each sorted table.

A variation on this can do any of the outer join algorithms.

See `https://toolz.readthedocs.io`; they offer `merge_sorted()`

Lookup Join

Greate for many small tables and one big table.
The "Star Schema" design pattern.

1. Transform each small table into a Python dictionary.

   ```
   small_1 = {r['pk']: row for row in table_1}
   ```

   etc.

2. Join.

   ```
   from_ = (
       (r, small_1[r['fk_1']], small_2[r['fk_2']])
       for r in big_table
   )
   ```

Key Point 2

You're not limited

You don't have to wrestle with database index and query optimizations.

You have the **Vast Python Ecosystem** available.

Key Point 2

You're not limited

You don't have to wrestle with database index and query optimizations.

You have the **Vast Python Ecosystem** available.

(Turn on echo effect)

Unlimited Computing Power!

Key Point 2

You're not limited

You don't have to wrestle with database index and query optimizations.

You have the **Vast Python Ecosystem** available.

(Turn on echo effect)

Unlimited Computing Power!

What about `Group by` and `Having`?
They can't be simple.

Group By Clause(s)

The **Group By** process involves two separate steps.

1. **Partition.** The expressions in the `GROUP BY` clause define keys to build groups.
2. **Aggregate.** The aggregate functions from the `SELECT` (and `HAVING`) are `reduce()` operations to create single group values.

Syntax Oddity: Group-By aggregates in the `SELECT` clause. And in the `HAVING` clause.

Group By Implementation

```
from collections import defaultdict
from operator import itemgetter
# Partition
groups = defaultdict(list)
for row_tuple in where:
    key = (k_1(row_tuple), k_2(row_tuple), ...)
    groups[key].append(row_tuple)
# Aggregate
group_by = []
for key, group in groups:
    agg_1 = some_function(group)
    agg_2 = mean(row['value'] for row in group)
    agg_3 = sum(map(itemgetter('name'), group))
    group_by.append((key, agg_1, agg_2, agg_3))
```

Having Clause

The **Having** process is (nearly) the same as the **Where** process. It's an expression to filter the groups.

SQL syntax uses aggregate functions in the `HAVING` clause.

- These are yet more group-by aggregates.
- The result values are only used for filtering.

Key Point 3

You're not limited

The Group-By operation is a `defaultdict(list)`.
Maybe a `Counter`.

You have the **Vast Python Ecosystem** available.

Key Point 3

You're not limited

The Group-By operation is a `defaultdict(list)`.
Maybe a `Counter`.

You have the **Vast Python Ecosystem** available.

(Turn on echo effect)

Unlimited Computing Power!

Conclusion

- SQL has helpful patterns to describe a desired result.
- SQL can be limiting.
- An actual database engine introduces a lot of overhead. Avoid it.

Think of SQL as a design language.

Conclusion

- SQL has helpful patterns to describe a desired result.
- SQL can be limiting.
- An actual database engine introduces a lot of overhead. Avoid it.

Think of SQL as a design language.
Not an implementation choice.

SQL

- SQL describes a pipeline of steps:

$$\text{From} \rightarrow \text{Where} \rightarrow \text{Select} \rightarrow \text{Group By} \rightarrow \text{Having}$$

- Or, nested functions:

$$H\left(G_a\left(G_p\left(S\left(W\left(F(t_1, t_2, ...)\right)\right)\right)\right)\right)$$

Important: The `select-from-where` ordering of clauses is confusing.
That's not how it works.

SQL to Python

In Python, `Select` is a stack of generator expressions

```
from_ = itertools.product(...)
where = filter(condition, from_)
select = map(row_builder, where)
groups = group_reduce(select)
aggregates = map(agg_row_builder, groups)
result = filter(having_condition, aggregates)
```

Most steps are lazy and don't compute big intermediate results.

SQL to Python

In Python, `Select` is a stack of generator expressions

```
from_ = itertools.product(...)
where = filter(condition, from_)
select = map(row_builder, where)
groups = group_reduce(select)
aggregates = map(agg_row_builder, groups)
result = filter(having_condition, aggregates)
```

Most steps are lazy and don't compute big intermediate results.
The `group_reduce()` function does compute a big result.

Call to Action

Stop using SQL (and a database) as a data transformation tool.

Continue using SQL as a design aid.

SQL design patterns are useful.

Look at SQL as a pipeline of functional transformations.

More Information

- *Unlearning SQL* book (Available from Amazon and Lulu)
- https://github.com/slott56/functional-SQL
- https://github.com/slott56/unlearning-sql
- https://fosstodon.org/@slott56