Analysis
July 7, 2016 • 8 minute read

Thinking in SQL vs Thinking in Python

BENN STANCIL
CHIEF ANALYST

Over the years, I've used a variety of languages and tools to analyze data. As I think back on my time using each tool, I've come to realize that each encourages a different mental framework for solving analytical problems. Being conscious of these frameworks—and the behaviors they promote—can be just as important as mastering the technical features of a new language or tool.

I was first introduced to data analysis about ten years ago as a college student (my time studying the backs of baseball cards notwithstanding). In school, and later as a economics researcher, I worked almost exclusively in two tools—Excel and R—

But when I started a new job focused on business analytics, I was thrown into a whole new world. The overwhelming majority of data I needed lived in databases. SQL became a necessary entry point.

At first, using a new tool was challenging. Not only was there a learning curve to contend with, but SQL felt limited. I was used to R's easy statistical tests, accessible plotting libraries, and flexible data structures. In SQL, creating something as seemingly simple as a box plot requires an entire blog post to learn.

For a while, I held on to this flexibility by cobbling the tools together, importing query results into RStudio.

Jumping from my SQL client to the R client made it hard to keep pieces of work together. The pain of downloading query results and reimporting them into R made my workflow linear—SQL to R, and never back again. If I wanted to add to an analysis, my first thought was usually, "Maybe I can find a way to do it with the data I already have." Knowing that this was how I'd feel once in R, I'd over-invest in a query by adding way more data than I initially needed. I was slow to find answers, and often left interesting lines of questioning unexplored.

Over time, as I got better at SQL, I started to appreciate its benefits. Its rigidity forced a consistency that was valuable for standardizing business metrics. Counts and simple aggregates—which SQL excels at—were often more than enough to inform business decisions. Eventually, I found myself giving up R in my workflow. Given how fragmented the SQL-to-R workflow was, the flexibility of R wasn't worth the switching cost. And when I really missed something in R, I could often find a long-winded workaround that mostly got the job done.

When the Mode team started working to integrate Python Notebooks with our SQL editor, I got pulled back into using a scripted language for analysis. This time though, the pain of switching tools—Mode passes SQL results to Python without any importing or exporting—was alleviated. Suddenly it felt like another world opened up again.

But with it came yet another way of thinking and a new mental framework to learn

SQL to R workflow without the costs—were clearly enough to make the transition worth it, it would've been much easier had I known a few things up front.

It's okay to not know all of the pieces

SQL is a pretty narrow language. Queries almost exclusively use some combination of joins, aggregate functions, subqueries, and window functions.

It's like building something with a set of basic Lego pieces. I can create amazing, complex, and original things, but I rarely uncover specialized pieces I never knew I had.

Python, by contrast, is like a collection of specialized Lego sets. Each library has its own custom pieces for building something very specific: Seaborn for visuals, pandas for analysis, scikit-learn for machine learning, and so on.

These libraries add a ton of power. What took dozens of lines of SQL (or wasn't possible at all) now requires just one Python method. But all these pieces no longer fit in my head. With SQL, if I go down a certain path, I might discover a better way to rearrange what I'm doing. In Python, I'm just as likely to find a new method for solving a problem as I am to discover a smarter way to rework the existing pieces.

Google it

The "Python community," which produces these specialized pieces, is a large and thriving group. The "SQL community" is... SEO fodder for Microsoft and Oracle. The cause for this is its own topic for debate, but the effect is clear—it's much easier to find genuinely supportive Python resources online.

Perhaps more importantly, the structure of Python lends itself to finding answers in ways that SQL struggles. Because Python has more pieces and it tends to be more abstracted from the data it's working on, people can easily share libraries and chunks of script. Someone sharing an example of how to create a boxplot can provide the code and say "Reference your data here." I could use a flight dataset to

learn how to create box plots with the pandas `.boxplot()` method and then turn around and use `.boxplot()` on app user data.

SQL queries, on the other hand, are intimately tied to data. I have to know the table and column names for the query to make any sense. This tends to makes sharing examples much harder. Showing someone how to make a box plot with SQL requires sharing data for it to make sense, which likely makes people more reluctant to share their work.

Embrace the black box

Give me enough time, and I can figure out what a SQL query is doing. Because all the pieces are typically familiar and because queries are self-contained entities that don't outsource work to external libraries, I can eventually piece together the full scope of the query and "get" it.

When creating or reviewing a Python notebook, sometimes I have to accept that I'll never fully know what's happening. To understand a SciPy function, for example, I'd have to go through piles of documentation or several layers of source code. Practically, that just doesn't happen. Instead, I more often find the operations that look like they apply, plug them in, and see if they return a result that passes my gut check.

This is the downside of having such a wide array of community resources and libraries. There's nothing necessarily wrong with it (and it's not like anyone, truly, fully understands what on earth `GROUP BY` really does), but with Python, I tend to find myself taking results a bit more on faith, especially when technical documentation is no easier to understand than the code itself.

Create webs of analysis

When working in SQL, queries often evolve linearly. I mostly use SQL to aggregate and join data from large data sets in a flow like this:

2. Aggregate this into something much smaller.
3. Join on more tables.
4. Aggregate again.
5. And so on, until I have a giant query that produces 12 rows of data.

Though these queries can be long, they often aren't hard to model mentally—data moves linearly up through subqueries. Data is like a snowball rolling downhill—it collects and compresses more tables as it goes, until it's transformed into a perfect snowball (or produces a catastrophic avalanche).

Python progresses more fluidly. I often take an initial dataset and break it into many smaller ones, and each becomes a thread of analysis. I can combine these threads together and pull them back apart in different ways.

This requires me to mentally model my analysis as a web, which is harder to do. I can no longer think, "What is my leftmost table? What am I going to join to it?" Instead, I have to think, "What's the best table to slice into lots of views? How can I stitch together those views into something meaningful?" As is the case with Python's "pieces," this “web” way of thinking provides a looser framework than SQL, which can be discomforting for people used to the tighter rails.

Skip around

The linear nature of SQL means that if I want to add some perspective to my analysis that's best introduced at the beginning, I have to back my way up my queries, and then pass that result all the way back down. It's cumbersome, even if it's as simple as adding a column through several layers of subqueries.

In other cases, pursuing different ideas in a single sitting requires creating several explorations of a core query. These independent queries become difficult to merge later—and compound the problems with changing things earlier in my analysis, as it'd require me to make that adjustment in all of these explorations.

The upside of Python's web-based workflow is that I can more easily skip around and iterate much faster. Things can be added and removed at different points, allowing me to pursue questions when they come up and in the same workflow. Since each analytical exploration can point to the same core, it's more natural to move between different points of analysis.

Also, often, it's just computationally faster. Nothing is more frustrating than writing a long-running SQL query only to find I wanted to add an additional column to the outermost query. Python allows me to apply this change directly to my last step, without having to rerun the entire operation each time through.

Welcome change

Ultimately, the biggest obstacle I've faced during this transition is inertia. SQL is my safe default. Even if the SQL query is ten times longer than the equivalent Python script, it feels easier to do it the way I already understand. Learning is harder than typing.

But clinging to what you know is penny-wise and pound foolish. Learning a new language not only provides new tools, but also opens up novel ways to think mentally model your data and analysis. This can lead you to ask and answer questions you would have never considered otherwise.

Recommended articles

- Bridge the Gap: Window Functions in Python and SQL
- Bridge the Gap: Python and SQL Overlap
- 10 Useful Python Data Visualization Libraries for Any Discipline
- How to Make the Leap from Excel to SQL
- The Python Tutorial for Data Analysis