Why aren't more data people talking about ibis?

Ian D. Gow

iandgow@gmail.com

4 December 2025

> **Note**
>
> This note is basically a copy-paste of a note by Joram Mutenge. The main change I make is to switch from PostgreSQL to DuckDB, as the setup costs of the latter are much lower.

If you started working in the data field 20 years ago, you probably used a lot of SQL. It's a robust, 50-year-old technology that excels at querying data, thanks to decades of research and development in database engines.

> **Note**
>
> I started in strategy consulting 30 years ago (not even "the data field") and I used SQL a lot back then. I wrote a little bit about that here.

But if you're working in data analyst or scientist today, you're likely more familiar with dataframe APIs like Pandas, Polars, or PySpark. Dataframe libraries have become the standard in data science and analytics. In fact, more data professionals today know how to use a dataframe library like Pandas than SQL. Currently, Pandas is the most widely adopted, but recently Polars has become so good it can't be ignored.

Like many people, I find dataframe libraries easier and more intuitive to work with than traditional SQL. Their syntax often feels more flexible and expressive, especially when working in Python-centric environments.

1 Downsides of dataframe libraries

While dataframe libraries like Pandas are convenient and user-friendly, they often lack the kind of optimized query engines that databases provide. SQL continues to shine when it comes to performance, particularly for operations like `GROUP BY` aggregations. Even Wes McKinney, the creator of Pandas, has acknowledged that complex aggregation operations in Pandas can be "awkward and slow."

Another downside is that writing new rows of data to a dataframe can be tedious and computationally expensive. This is largely because most dataframe libraries are columnar in design, whereas traditional database tables are row-oriented and optimized for such operations.

2 Combining database and dataframe APIs

What if we could combine the elegance of dataframe syntax with the efficiency of database engines? That would allow us to write powerful queries in a syntax that's accessible to both SQL pros and their

colleagues who may not be familiar with SQL, while still leveraging the power of a backend database engine to perform query optimizations.

If only there were a project that brought together the best of both worlds. Well, there is—and it's called Ibis. It's like a match made in heaven!

3 What is Ibis?

Ibis is a Python library that provides a high-level, dataframe-like interface for working with structured data across different backends, such as SQL databases, big data engines like Apache Spark, and in-memory dataframes like Pandas or Polars. It allows users to write expressive and chainable code for data manipulation and analysis without worrying about the backend engine processing the data. This means you can efficiently run your queries with the same syntax on different systems.

If you've worked with different SQL databases, you've likely noticed that each one can have its own SQL dialect. This can make it hard to scale your workflow from local development to large production environments. Ibis solves this problem by letting you use the same code across multiple databases.

Wes McKinney created Ibis, which makes me trust that it's a great project. Why? Because he learned from the mistakes of Pandas which he outlined in his 10 Things I Hate About pandas blog post, and he's correcting those mistakes with Ibis.

4 Business reasons to use Ibis

You may be asking, "Why should my business switch to Ibis?" Beyond being a cool piece of technology, there are solid business reasons to adopt Ibis in your company. Here are two good reasons.

1. **It acts as an insurance policy** - If your core vendor changes their policy or switches to a different database, you won't need to rewrite your code. Just update the backend, and your existing code will continue to run smoothly. This can save you tens–or even hundreds–of developer hours.
2. **It prevents vendor lock-in** - At some point, you'll probably want to move your data out of a vendor system and use it elsewhere—like in a browser for visualization or in a dashboard. With Ibis, you don't need to write custom code for each database. The same Ibis code works whether you're using Microsoft SQL Server, MySQL, or another system. Once again, this saves your company time and money.

5 Setting up Ibis

We've talked about how great Ibis is, but how do you actually use it to answer questions about your data? I'll walk you through setting up the database, loading the data, and finally analyzing it. You'll see just how easy it is to answer business questions using Ibis queries.

5.1 Installing the database

> **Note**
>
> The original note used PostgreSQL, while I use DuckDB. So there's not much of a database setup step in this version. I believe `pip install ibis` in the original note installs a different package from the Ibis we care about, so I just use the line below.

For this walkthrough, we're going to download DuckDB and install it on our machine. I recommend using the same approach to make it easy to follow along.

```
pip install 'ibis-framework[duckdb]'
```

6 Getting the data into a database

We'll use a dataset from a clothing store, which you can download here. The dataset contains clothing sales data over two years. It has 8 columns and 1,000 rows. If you download it, you'll notice that it's a CSV file I like this dataset because it's rich enough to help us explore the capabilities of Ibis.

> **Note**
>
> To minimize differences I retain the column names used in the `CREATE TABLE` statement in the original note. However, I make these column names lower case; the original note relied on the (qualified) "case indifference" of PostgreSQL, as I think it's not a good idea to ignore case in a Python context.

```
import ibis
from ibis import _

con = ibis.duckdb.connect()

raw_url = (
    "https://raw.githubusercontent.com/jorammutenge/learn-rust/main/"
    "sample_sales.csv")

t = con.read_csv(
    raw_url,
    table_name="sales_orders",
    header=False,
    skip=1,
    names=["account_num", "account_name", "sku", "category", "quantity",
           "unit_price", "ext_price", "date"])
```

> **Note**
>
> We can check that the data are in `sales_table`, but below I will just use `t` to access the data.

```
con.table('sales_orders')
```

```
┏━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━
┃ account_num ┃ account_name                        ┃ sku      ┃ category ┃ quantity ┃
┃ unit_price ┃ ext_price ┃ date                 ┃
┡━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━
│ int64       │ string                              │ string   │ string   │ int64    │
float64     │ float64     │ timestamp(6)         │
├─────────────┼─────────────────────────────────────┼──────────┼──────────┼──────────┼─────
│      803666 │ Fritsch-Glover                      │ HX-24728 │ Hat      │        1
│      98.98 │      98.98 │ 2014-09-28 11:56:02 │
```

```
│       64898 │ O'Conner Inc                       │ LK-02338 │ Sweater  │        9
│      34.80 │    313.20 │ 2014-04-24 16:51:22 │
│      423621 │ Beatty and Sons                    │ ZC-07383 │ Sweater  │       12
│      60.24 │    722.88 │ 2014-09-17 17:26:22 │
│      137865 │ Gleason, Bogisich and Franecki     │ QS-76400 │ Sweater  │        5
│      15.25 │     76.25 │ 2014-01-30 07:34:02 │
│      435433 │ Morissette-Heathcote               │ RU-25060 │ Sweater  │       19
│      51.83 │    984.77 │ 2014-08-24 06:18:12 │
│      198887 │ Shanahan-Bartoletti                │ FT-50146 │ Sweater  │        4
│      18.51 │     74.04 │ 2014-09-05 07:24:23 │
│      969663 │ Gusikowski, Reichert and Gerlach   │ AE-95093 │ Socks    │        4
│      49.95 │    199.80 │ 2014-04-28 21:51:24 │
│        1288 │ Wilderman, Herman and Breitenberg  │ FT-50146 │ Sweater  │       14
│      68.20 │    954.80 │ 2013-12-04 13:53:26 │
│      979589 │ Brown Inc                          │ HX-24728 │ Hat      │       16
│      52.99 │    847.84 │ 2014-02-07 14:53:59 │
│      839884 │ Turcotte, Turner and Anderson      │ FT-50146 │ Sweater  │        8
│      21.35 │    170.80 │ 2014-09-03 16:06:44 │
│           … │ …                                  │ …        │ …        │        …
│          … │         … │ …                    │
└─────────────┴────────────────────────────────────┴──────────┴──────────┴──────────┴
```

While we're at it, let's verify the backend that our queries will be using.

```
t.get_backend()
```

```
<ibis.backends.duckdb.Backend at 0x118e4f250>
```

7 Analyzing data with Ibis

We can now begin answering business questions using our dataset. It's important to note that while we write our queries using Ibis in a dataframe-like syntax, the actual data processing is handled by DuckDB. We're leveraging the power of DuckDB's query engine to perform the computations.

7.1 Table transformation

The table displayed above shows the date as the last column. I prefer to have the date as the first column in my datasets. Let's make that change.

```
t = t.relocate('date')
t
```

```
┏━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━
┃ date               ┃ account_num ┃ account_name                       ┃ sku      ┃
category ┃ quantity ┃ unit_price ┃ ext_price ┃
┡━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━
│ timestamp(6)       │ int64      │ string                             │ string   │
string   │ int64    │ float64    │ float64   │
├────────────────────┼────────────┼────────────────────────────────────┼──────────┼──────────┼─
```

```
| 2014-09-28 11:56:02 |      803666 | Fritsch-Glover                    | HX-24728
| Hat      |        1 |      98.98 |     98.98 |
| 2014-04-24 16:51:22 |       64898 | O'Conner Inc                      | LK-02338
| Sweater  |        9 |      34.80 |    313.20 |
| 2014-09-17 17:26:22 |      423621 | Beatty and Sons                   | ZC-07383
| Sweater  |       12 |      60.24 |    722.88 |
| 2014-01-30 07:34:02 |      137865 | Gleason, Bogisich and Franecki    | QS-76400
| Sweater  |        5 |      15.25 |     76.25 |
| 2014-08-24 06:18:12 |      435433 | Morissette-Heathcote              | RU-25060
| Sweater  |       19 |      51.83 |    984.77 |
| 2014-09-05 07:24:23 |      198887 | Shanahan-Bartoletti               | FT-50146
| Sweater  |        4 |      18.51 |     74.04 |
| 2014-04-28 21:51:24 |      969663 | Gusikowski, Reichert and Gerlach  | AE-95093
| Socks    |        4 |      49.95 |    199.80 |
| 2013-12-04 13:53:26 |        1288 | Wilderman, Herman and Breitenberg | FT-50146
| Sweater  |       14 |      68.20 |    954.80 |
| 2014-02-07 14:53:59 |      979589 | Brown Inc                         | HX-24728
| Hat      |       16 |      52.99 |    847.84 |
| 2014-09-03 16:06:44 |      839884 | Turcotte, Turner and Anderson     | FT-50146
| Sweater  |        8 |      21.35 |    170.80 |
| …                   |          … | …                                 | …
…        |        … |          … |         … |
```

> **Note**
>
> While the `t` object has been modified by `relocate()`, the underlying table in DuckDB remains unchanged.

```
con.table('sales_orders')
```

```
┌─────────────┬──────────────────────────────────┬──────────┬──────────┬──────────┬──────────┬──────────┬──────────┐
│ account_num │ account_name                     │ sku      │ category │ quantity │ unit_price │ ext_price │ date                │
├─────────────┼──────────────────────────────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┤
│ int64       │ string                           │ string   │ string   │ int64    │ float64    │ float64   │ timestamp(6)        │
├─────────────┼──────────────────────────────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┤
│      803666 │ Fritsch-Glover                   │ HX-24728 │ Hat      │        1 │    98.98 │    98.98 │ 2014-09-28 11:56:02 │
│       64898 │ O'Conner Inc                     │ LK-02338 │ Sweater  │        9 │    34.80 │   313.20 │ 2014-04-24 16:51:22 │
│      423621 │ Beatty and Sons                  │ ZC-07383 │ Sweater  │       12 │    60.24 │   722.88 │ 2014-09-17 17:26:22 │
│      137865 │ Gleason, Bogisich and Franecki   │ QS-76400 │ Sweater  │        5 │    15.25 │    76.25 │ 2014-01-30 07:34:02 │
│      435433 │ Morissette-Heathcote             │ RU-25060 │ Sweater  │       19 │    51.83 │   984.77 │ 2014-08-24 06:18:12 │
```

```
│      198887 │ Shanahan-Bartoletti                │ FT-50146 │ Sweater  │        4
│      18.51 │     74.04 │ 2014-09-05 07:24:23 │
│      969663 │ Gusikowski, Reichert and Gerlach  │ AE-95093 │ Socks    │        4
│      49.95 │    199.80 │ 2014-04-28 21:51:24 │
│        1288 │ Wilderman, Herman and Breitenberg │ FT-50146 │ Sweater  │       14
│      68.20 │    954.80 │ 2013-12-04 13:53:26 │
│      979589 │ Brown Inc                         │ HX-24728 │ Hat      │       16
│      52.99 │    847.84 │ 2014-02-07 14:53:59 │
│      839884 │ Turcotte, Turner and Anderson     │ FT-50146 │ Sweater  │        8
│      21.35 │    170.80 │ 2014-09-03 16:06:44 │
│           … │ …                                 │ …        │ …        │        …
│          … │         … │ …                   │
└─────────────┴───────────────────────────────────┴──────────┴──────────┴──────────┴──────────────┘
```

8 Aggregations and sorting

Let's determine which clothing category generates the most revenue, and then rank all the clothing categories in descending order based on their total revenue.

```
(t
 .group_by('category')
 .aggregate(ext_price=_.ext_price.sum())
 .order_by(_.ext_price.desc())
 .execute()
 )
```

	category	ext_price
0	Sweater	301716.00
1	Socks	169169.93
2	Hat	99294.01

What if we wanted to know what the top-selling SKUs are by quantity and revenue and then sort the values by quantity in ascending order.

```
(t
 .group_by('sku')
 .agg(quantity=_.quantity.sum(),
      ext_price=_.ext_price.sum())
 .order_by(_.quantity.asc())
 )
```

```
┏━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━━━┓
┃ sku      ┃ quantity ┃ ext_price  ┃
┡━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━━━┩
│ string   │ int64    │ float64    │
├──────────┼──────────┼────────────┤
│ HX-24728 │      756 │  39909.44 │
```

```
| GS-95639 |      823 |  44710.44 |
| FT-50146 |      995 |  53558.69 |
| LK-02338 |     1036 |  54329.78 |
| AE-95093 |     1062 |  57146.97 |
| XX-25746 |     1133 |  59384.57 |
| ZC-07383 |     1144 |  61483.76 |
| RU-25060 |     1167 |  62957.70 |
| QS-76400 |     1209 |  69386.07 |
| IC-59308 |     1240 |  67312.52 |
└──────────┴──────────┴───────────┘
```

Just for fun, let's perform multiple aggregates in a single query. We'll calculate the total, average, and median sales value for each day of the week, then sort the results by average sales.

This query includes creating a new column called day based on the date column.

```
(t
 .mutate(day=_.date.day_of_week.full_name())
 .group_by('day')
 .agg(total_sales=_.ext_price.sum(),
      avg_sales=_.ext_price.mean(),
      median_sales=_.ext_price.median(),)
 .order_by(_.avg_sales.desc())
 )
```

```
┏━━━━━━━━━━━┳━━━━━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━┓
┃ day       ┃ total_sales ┃ avg_sales  ┃ median_sales ┃
┡━━━━━━━━━━━╇━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━┩
│ string    │ float64     │ float64    │ float64      │
├───────────┼─────────────┼────────────┼──────────────┤
│ Sunday    │    91575.29 │ 627.228014 │      560.825 │
│ Saturday  │    81985.61 │ 611.832910 │      512.260 │
│ Monday    │    88708.66 │ 591.391067 │      411.620 │
│ Tuesday   │    85035.91 │ 578.475578 │      470.890 │
│ Wednesday │    83444.28 │ 563.812703 │      454.200 │
│ Friday    │    69183.76 │ 528.120305 │      456.040 │
│ Thursday  │    70246.43 │ 487.822431 │      395.145 │
└───────────┴─────────────┴────────────┴──────────────┘
```

9 Filtering and creating columns

We want to know which year had the highest revenue and what was the dollar value of that revenue. Here's the query to do just that.

```
(t
 .mutate(year=_.date.year())
 .group_by('year')
 .agg(total_sales=_.ext_price.sum())
```

```
 .filter(_.total_sales == _.total_sales.max())
 )
```

```
┏━━━━━━━┳━━━━━━━━━━━━━┓
┃ year  ┃ total_sales ┃
┡━━━━━━━╇━━━━━━━━━━━━━┩
│ int32 │ float64     │
├───────┼─────────────┤
│  2014 │    425348.0 │
└───────┴─────────────┘
```

Suppose we want to identify the hours of the day with the least store traffic so we can reduce the number of employees during those times. After all, we don't want them standing around with nothing to do! To achieve this, we'll select the top three hours of the day with the lowest sales.

```
(t
 .mutate(hour=_.date.hour())
 .group_by('hour')
 .agg(total_sales=_.ext_price.sum())
 .order_by(_.total_sales.asc())
 .limit(3)
 )
```

```
┏━━━━━━━┳━━━━━━━━━━━━━┓
┃ hour  ┃ total_sales ┃
┡━━━━━━━╇━━━━━━━━━━━━━┩
│ int32 │ float64     │
├───────┼─────────────┤
│    16 │    13706.50 │
│    22 │    16160.71 │
│    21 │    16329.77 │
└───────┴─────────────┘
```

It's winter, and we'd like to identify all customers who purchased either socks or a sweater. We plan to send them an email about our new holiday-themed socks and sweaters now in stock.

```
(t
 .select('account_name', 'category')
 .filter(_.category.isin(['Sweater', 'Socks']))
 .select('account_name')
 .distinct()
)
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ account_name              ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│ string                    │
```

```
├──────────────────────────────┤
│ Dickinson-McGlynn            │
│ Corkery, King and Cassin     │
│ Zieme-Romaguera              │
│ Bartell, Cronin and Abbott   │
│ Bartoletti Inc               │
│ Wilkinson, Pagac and Cole    │
│ Leannon-Langosh              │
│ Weimann-Parisian             │
│ Hickle Ltd                   │
│ Maggio Ltd                   │
│ …                            │
└──────────────────────────────┘
```

Next, let's find customers who bought **both** socks and sweaters. These loyal shoppers will receive a special 5% discount as a thank-you for their continued support.

```
(t
 .filter(_.category.isin(['Sweater', 'Socks']))
 .group_by('account_name')
 .having(_.category.nunique() == 2)
 .select('account_name')
)
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ account_name                         ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│ string                               │
├──────────────────────────────────────┤
│ O'Conner Inc                         │
│ Beatty and Sons                      │
│ Gleason, Bogisich and Franecki       │
│ Morissette-Heathcote                 │
│ Shanahan-Bartoletti                  │
│ Gusikowski, Reichert and Gerlach     │
│ Wilderman, Herman and Breitenberg    │
│ Turcotte, Turner and Anderson        │
│ Armstrong, Champlin and Ratke        │
│ Goyette, Kessler and Goodwin         │
│ …                                    │
└──────────────────────────────────────┘
```

10 Graph plotting

Yes, Ibis supports native plotting using the Altair visualization library. You can install it by running `pip install altair` in your command line or terminal.

Other popular visualization libraries like Plotly, Matplotlib, or Seaborn can also be used. However, you'll need to convert your Ibis tables to Pandas using the `.to_pandas()` function before using them with these libraries.

Let's create a bar chart that displays the total quantity sold for each product SKU to identify which SKU has the highest sales by quantity. More importantly, we'll arrange the bars from tallest to shortest. We'll use Altair to build the chart.

```
import altair as alt

bar_chart = (alt
 .Chart(t.group_by('sku').agg(total_quantity=_.quantity.sum()))
 .mark_bar(color='#556B2F')
 .encode(x=alt.X('sku', axis=alt.Axis(labelAngle=0, labelFontSize=14),
                 title=None,
                 sort=alt.EncodingSortField(field='total_quantity',
                                            order='descending')),
         y=alt.Y('total_quantity', axis=alt.Axis(labelFontSize=14), title=None),
         tooltip=['sku','total_quantity'])
 .properties(width=1000, height=500,
             title={'text':'Total quantity by product SKU', 'fontSize': 20})
 .configure(background='#FFE4B5')
 .interactive()
 )
bar_chart.show()
```

```
alt.Chart(...)
```

11 Pivot tables

If you typically do your data analysis in Excel, then pivot tables are likely your go-to tool. Fortunately, Ibis also supports pivot table transformations, so you won't need to switch back to Excel.

Let's create a pivot table that shows the total quantity for each month in 2014. The months will be the columns, with the total quantity values displayed in the corresponding column for each month.

```
(t
 .filter(t.date.year() == 2014)
 .mutate(month=t.date.strftime('%b-%Y'))
 .select('quantity','month')
 .pivot_wider(names_from='month', values_from='quantity',
              names_sort=True, values_agg='sum')
)
```

```
┏━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┓
┃ Apr-2014 ┃ Aug-2014 ┃ Feb-2014 ┃ Jan-2014 ┃ Jul-2014 ┃ Jun-2014 ┃ Mar-2014 ┃
May-2014 ┃ Sep-2014 ┃
┡━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━┩
│ int64    │ int64    │ int64    │ int64    │ int64    │ int64    │ int64    │ int64
│ int64    │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────
│      887 │     1065 │      935 │      833 │     1006 │      684 │      791 │
```

```
787 |      911 |
└──────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──────────
```

There's an issue with the result of the query above. It's not with the values themselves, but with how the columns are arranged. The months aren't in the correct calendar order. Let's fix that!

> **Note**
>
> The original note took a more complicated approach than the one I use here. I drop `names_sort=True` from `pivot_wider()` and instead create a new column `month` that truncates the date to the month, and then use that column to sort the data before formatting `month` as `'%b-%Y'`. I believe that the original code would not be robust to including data from different years, but the approach below would be.

```
(t
 .filter(t.date.year() == 2014)
 .mutate(month = t.date.truncate("month"))
 .order_by('month')
 .mutate(month=t.date.strftime('%b-%Y'))
 .select('quantity','month')
 .pivot_wider(names_from='month', values_from='quantity', values_agg='sum')
)
```

```
┏━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━┳━━━━━━━━━━
┃ Jan-2014 ┃ Feb-2014 ┃ Mar-2014 ┃ Apr-2014 ┃ May-2014 ┃ Jun-2014 ┃ Jul-2014 ┃
Aug-2014 ┃ Sep-2014 ┃
┡━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━╇━━━━━━━━━━
│ int64    │ int64    │ int64    │ int64    │ int64    │ int64    │ int64    │ int64
│ int64    │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────
│      833 │      935 │      791 │      887 │      787 │      684 │     1006 │
1065 │      911 │
└──────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──────────
```

11.1 Customer segmentation

We want to offer a discount to our loyal customers, which we define as any customer who has purchased clothing from every product category available in our store.

First, let's take a look at all the product categories we have in stock.

```
(t
 .select('category')
 .distinct()
 )
```

```
┏━━━━━━━━━━┓
┃ category ┃
```

```
┌──────────┐
│ string   │
├──────────┤
│ Sweater  │
│ Socks    │
│ Hat      │
└──────────┘
```

Since we know there are three product categories in stock, we can easily write a query to retrieve the list of loyal customers as follows:

```
(t
 .group_by('account_name')
 .aggregate(category_num=t.category.nunique())
 .filter(_.category_num == 3)
)
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━┓
┃ account_name                   ┃ category_num ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━┩
│ string                         │ int64        │
├────────────────────────────────┼──────────────┤
│ Upton, Runolfsson and O'Reilly │            3 │
│ Kuvalis-Roberts                │            3 │
│ Fritsch-Glover                 │            3 │
│ Mills Inc                      │            3 │
│ Herman Ltd                     │            3 │
│ Ledner-Kling                   │            3 │
│ Beier-Bosco                    │            3 │
│ Bashirian, Beier and Watsica   │            3 │
│ Koepp-McLaughlin               │            3 │
│ Halvorson PLC                  │            3 │
│ …                              │            … │
└────────────────────────────────┴──────────────┘
```

In most cases, we won't know in advance how many product categories are available. That means the previous query wouldn't be suitable. Here's the revised version of the query for situations where the number of product categories is unknown.

And since we're only interested in the customers, we'll remove `category_num` from the table.

```
(t
 .group_by('account_name')
 .aggregate(category_num=t.category.nunique())
 .filter(_.category_num == t.select(t.category).distinct().count())
 .select('account_name')
)
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ account_name                    ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│ string                          │
├─────────────────────────────────┤
│ Mills Inc                       │
│ Upton, Runolfsson and O'Reilly  │
│ Bashirian, Beier and Watsica    │
│ Beier-Bosco                     │
│ Herman Ltd                      │
│ Halvorson PLC                   │
│ Kuvalis-Roberts                 │
│ Fritsch-Glover                  │
│ Ledner-Kling                    │
│ Schultz Group                   │
│ …                               │
└─────────────────────────────────┘
```

To get the total number of loyal customers, we just need to add one line to the query.

```
(t
 .group_by('account_name')
 .aggregate(category_num=t.category.nunique())
 .filter(_.category_num == t.select(t.category).distinct().count())
 .count()
)
```

```
┌────┐
│ 11 │
└────┘
```

Now we've identified the 11 individual loyal customers eligible for the 10% discount. They'll be thrilled to receive this discount—and maybe even encouraged to make more purchases from our store.

Next, let's identify the top five customers who contribute the highest percentage of revenue to our business.

```
(t
 .group_by('account_name')
 .agg(ext_price=_.ext_price.sum())
 .mutate(percent=(_.ext_price / _.ext_price.sum()) * 100)
 .order_by(_.percent.desc())
 .head(5)
 )
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━┓
┃ account_name             ┃ ext_price ┃ percent  ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━┩
│ string                   │ float64   │ float64  │
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━┓
┃ D'Amore PLC               ┃   3529.41  ┃ 0.618999   ┃
┃ Wilderman Group           ┃   3466.92  ┃ 0.608040   ┃
┃ Hilll, Schultz and Braun  ┃   3390.29  ┃ 0.594600   ┃
┃ Kuvalis-Roberts           ┃   3296.00  ┃ 0.578063   ┃
┃ Mills Inc                 ┃   2852.69  ┃ 0.500314   ┃
┗━━━━━━━━━━━━━━━━━━━━━━━━━━━┻━━━━━━━━━━━━┻━━━━━━━━━━━━┛
```

The low revenue percentages from these top five customers suggest that our business is well-diversified. Losing any single customer would not have a major impact on our overall performance.

11.2 Saving data

Let's save the data for the top 5 customers in a CSV file, which we can share with the clothing store's shareholders. Ibis makes it easy to export data in multiple formats, including CSV, JSON, Excel, Parquet, and more.

```
(t
 .group_by('account_name')
 .agg(ext_price=_.ext_price.sum())
 .mutate(percent=(_.ext_price / _.ext_price.sum()) * 100)
 .order_by(_.percent.desc())
 .head(5)
 .to_csv('top_5_customers.csv')
 )
```

Here's the content of the CSV file saved to disk.

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┓
┃ account_name              ┃ ext_price          ┃ percent           ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
│ D'Amore PLC               │ 3529.4100000000003 │ 0.6189993285277622 │
│ Wilderman Group           │ 3466.92            │ 0.6080396304366649 │
│ Hilll, Schultz and Braun  │ 3390.29            │ 0.5946000134624163 │
│ Kuvalis-Roberts           │ 3296.0             │ 0.5780631286326905 │
│ Mills Inc                 │ 2852.69            │ 0.5003139885980552 │
└━━━━━━━━━━━━━━━━━━━━━━━━━━━┴━━━━━━━━━━━━━━━━━━━━┴━━━━━━━━━━━━━━━━━━━┘
```

12 Conclusion

We've only scratched the surface of what Ibis can do. Hopefully, I've whetted your appetite to give this powerful library a try because after using it, I think I've found the answer to the question I posed in the title. Maybe data people are not talking about Ibis because they're keeping it a secret. They want to be the only ones benefiting from it.

Well, the ibis is out of the nest now, and it's taking flight. Let's spread the word and let it soar across the data landscape. And to all the data podcasts out there, I'd be thrilled to be a guest on your show to talk about this unsung hero of a Python library in the data field. Do reach out!