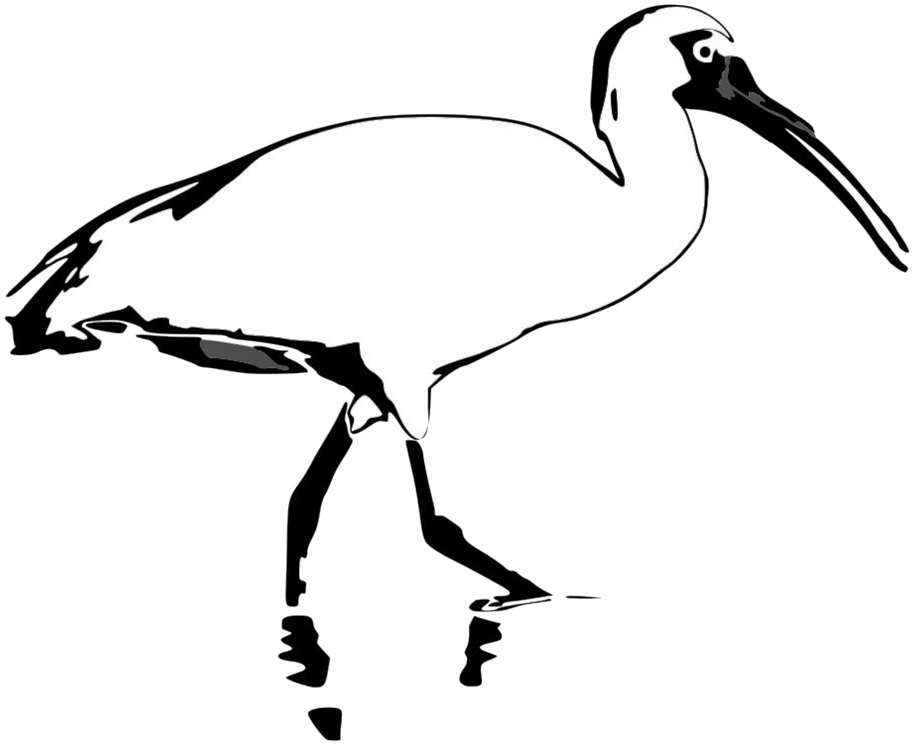


Ibis

Because SQL is everywhere
and so is Python

Intro

Gil Forsyth
Voltron Data

gforsyth

@gforsyth@fosstodon.org

Phillip Cloud
Voltron Data

cpcloud

Phillip in the Cloud
cpcloud

Intro

Phillip Cloud
Voltron Data

cpcloud

Phillip in the Cloud
cpcloud

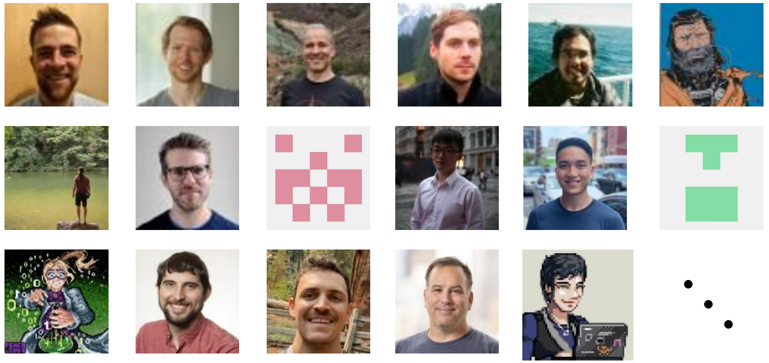
Gil Forsyth
Voltron Data

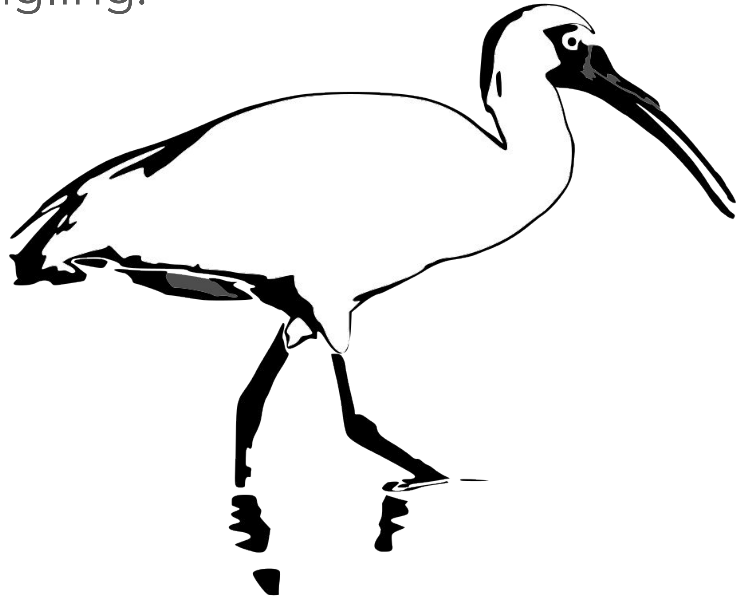
gforsyth

@gforsyth@fosstodon.org

Ibis!

A lightweight Python library for data wrangling.





Show of hands

- Translated data analysis from Pandas to PySpark?
- Prototyped something in Pandas then throw over the wall to a data engineer?
- Received some Pandas code that was thrown over a wall?
- Used parquet as a cross-language serialization format?

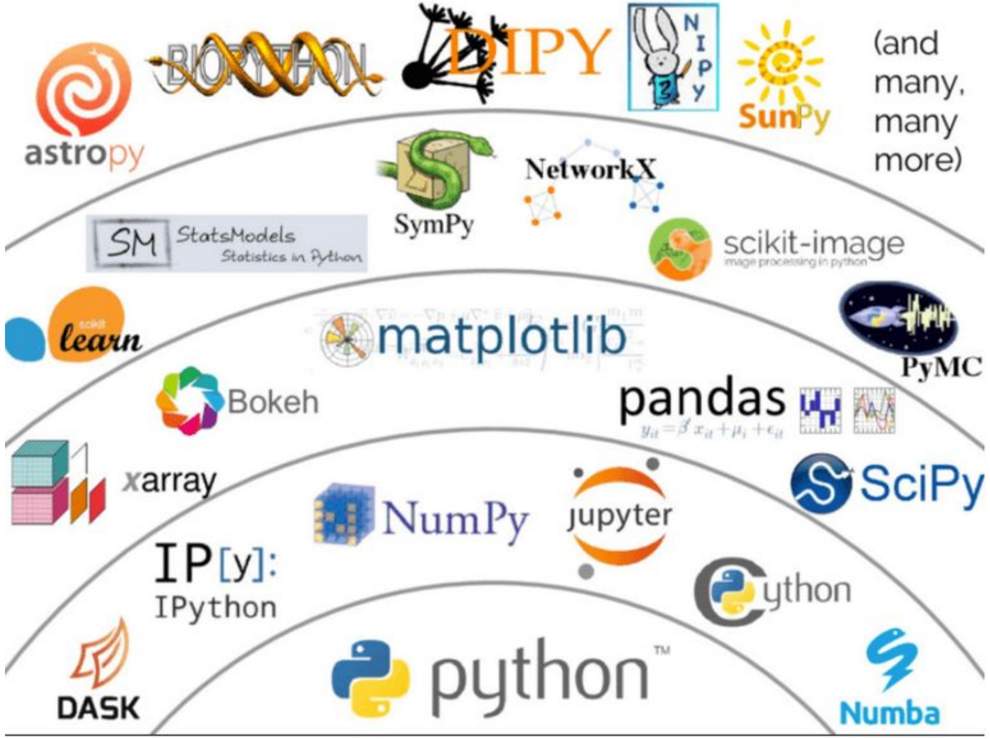
You’ve probably done at least one

- Translated code from Pandas to PySpark
- Prototyped in Pandas and thrown over the wall to data eng
- Been the data engineer on the other side of that wall
- Used parquet as a cross-language serialization format

The PyData Stack

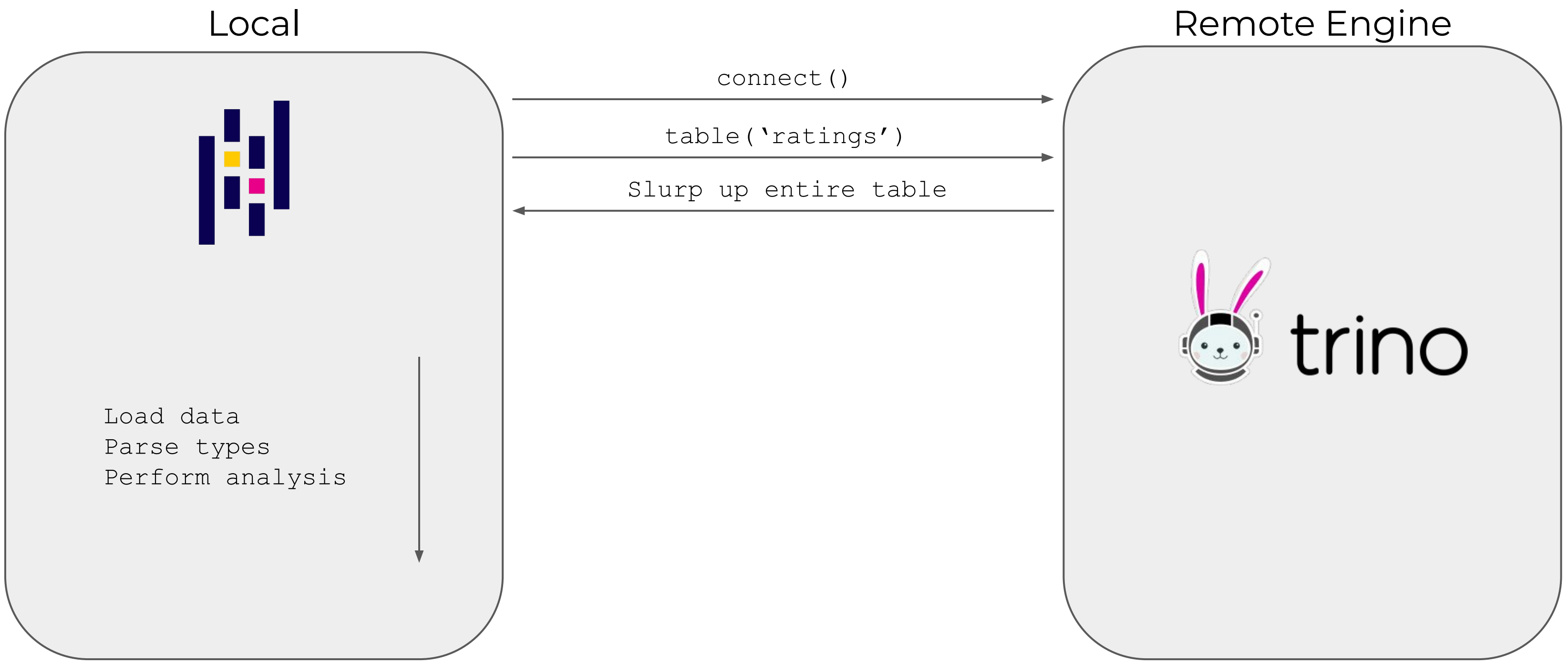
Data is local

Data fits in memory



Adapted from Jake VanderPlas, “The Unreasonable Effectiveness of Python in Science”, PyCon 2017

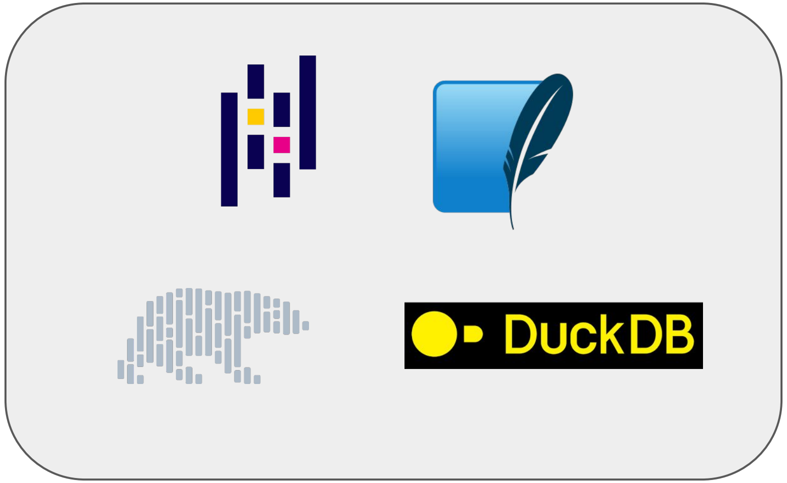
Local Execution



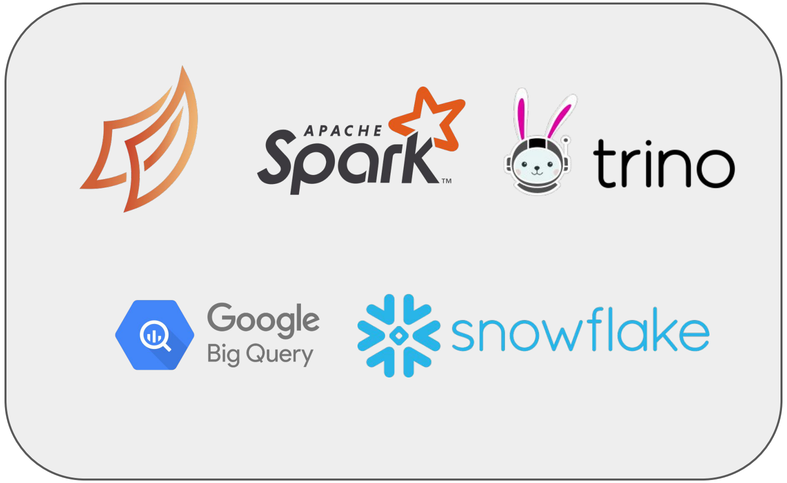
The PyData translation problem

No one *wants* to write things twice, but...

Local / Dev



Distributed / Prod



We need to talk about SQL

It’s EVERYWHERE

And it’s between you and the data.

trino

snowflake

druid

MariaDB

APACHE Spark™

Google Big Query

SQL

Pros

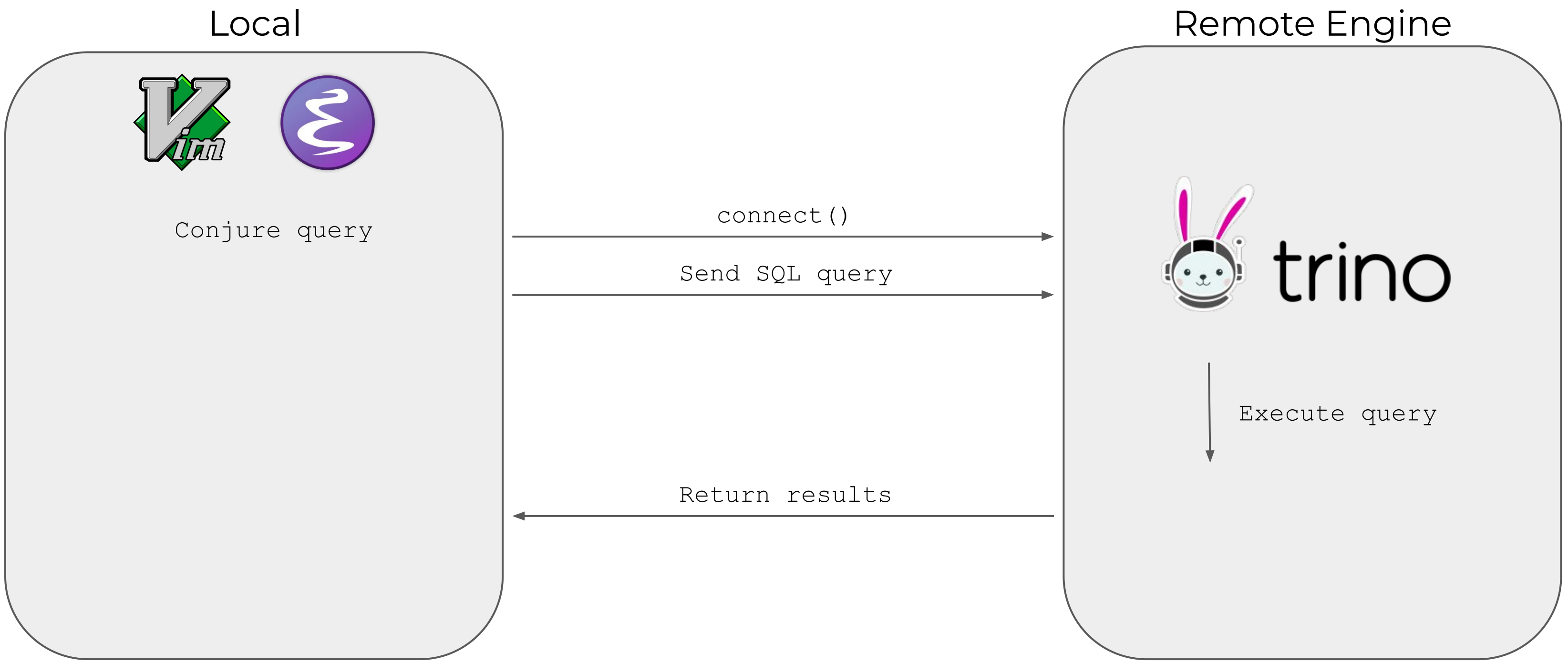
- Standardized†
- Concise*

Cons

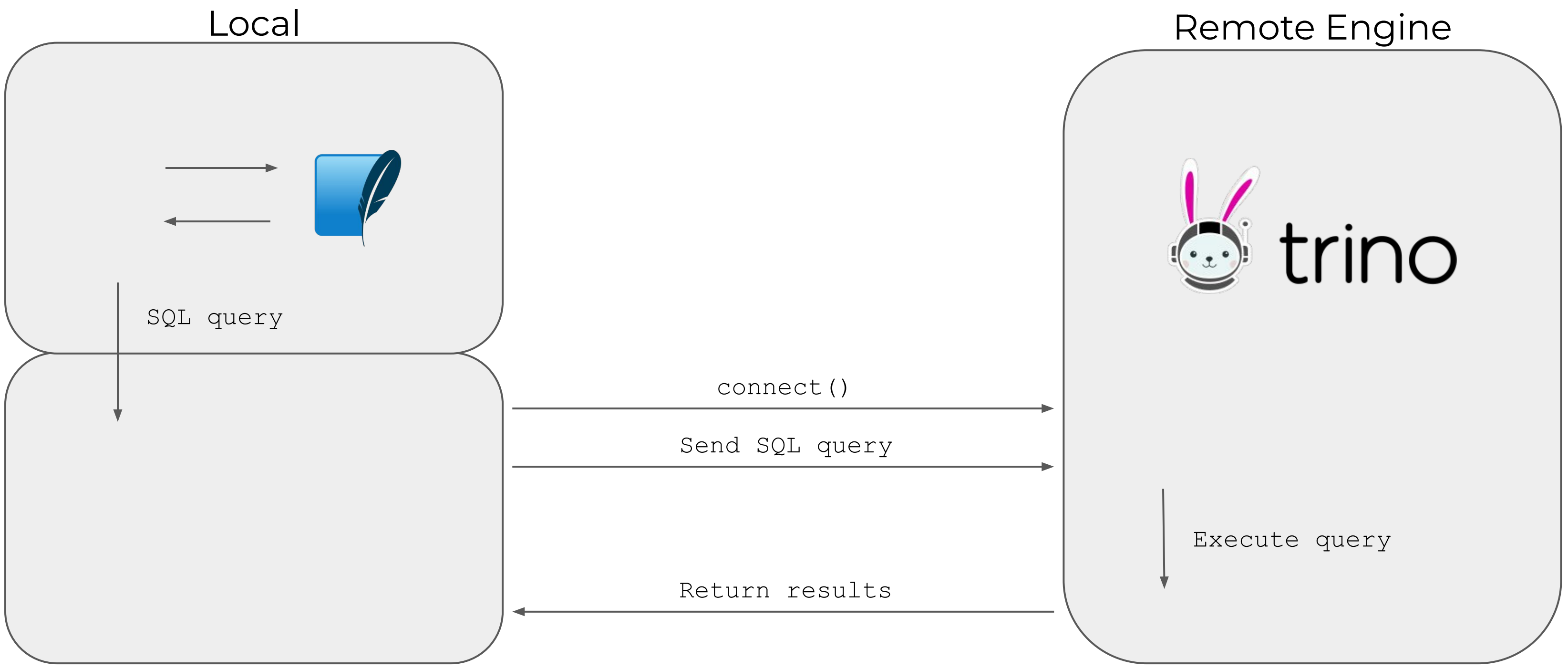
- Effectively untestable*
- *: Sometimes inscrutable
- Slow feedback

†: kind of, but also not really

Remote Execution (the good kind)



Remote Execution (the good kind)

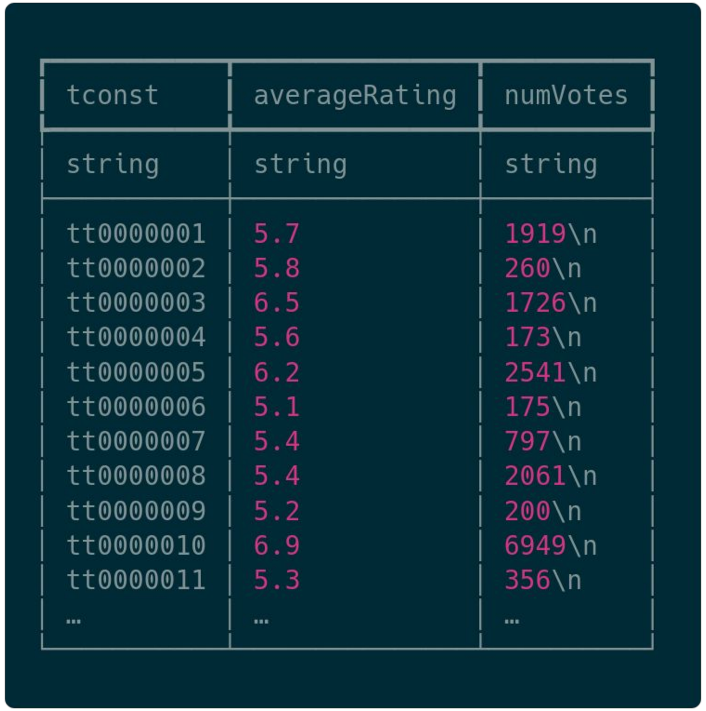


Problem solved*.

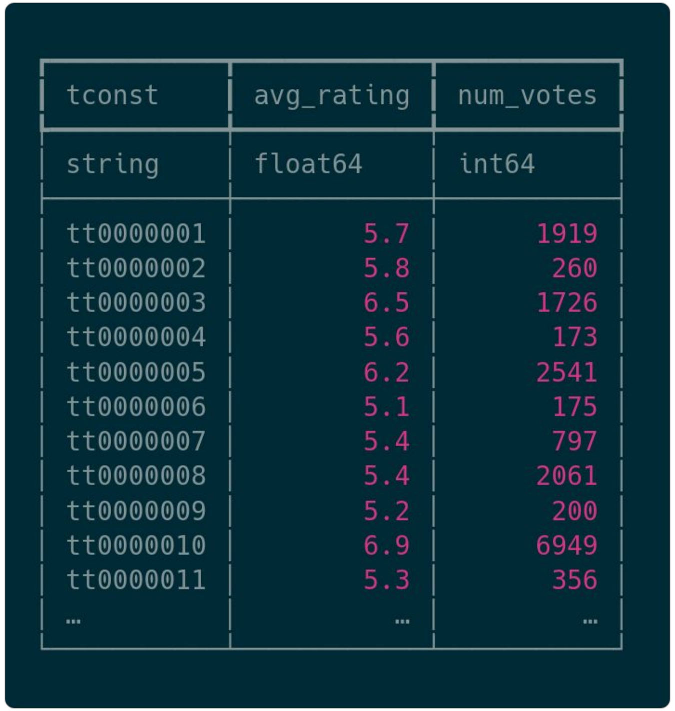
*Narrator: It was not

The translation problem

The SQL standard is a standard but how standard are standards?



```
┌───────────┬───────────────┬──────────┐
│ tconst    ┆ averageRating ┆ numVotes │
│ ---       ┆ ---           ┆ ---      │
│ string    ┆ string        ┆ string   │
╞═══════════╪═══════════════╪══════════╡
│ tt0000001 ┆ 5.7           ┆ 1919\n   │
│ tt0000002 ┆ 5.8           ┆ 260\n    │
│ tt0000003 ┆ 6.5           ┆ 1726\n   │
│ tt0000004 ┆ 5.6           ┆ 173\n    │
│ tt0000005 ┆ 6.2           ┆ 2541\n   │
│ tt0000006 ┆ 5.1           ┆ 175\n    │
│ tt0000007 ┆ 5.4           ┆ 797\n    │
│ tt0000008 ┆ 5.4           ┆ 2061\n   │
│ tt0000009 ┆ 5.2           ┆ 200\n    │
│ tt0000010 ┆ 6.9           ┆ 6949\n   │
│ tt0000011 ┆ 5.3           ┆ 356\n    │
│ …         ┆ …             ┆ …        │
└───────────┴───────────────┴──────────┘
```



```
┌───────────┬────────────┬───────────┐
│ tconst    ┆ avg_rating ┆ num_votes │
│ ---       ┆ ---        ┆ ---       │
│ string    ┆ float64    ┆ int64     │
╞═══════════╪════════════╪═══════════╡
│ tt0000001 ┆        5.7 ┆      1919 │
│ tt0000002 ┆        5.8 ┆       260 │
│ tt0000003 ┆        6.5 ┆      1726 │
│ tt0000004 ┆        5.6 ┆       173 │
│ tt0000005 ┆        6.2 ┆      2541 │
│ tt0000006 ┆        5.1 ┆       175 │
│ tt0000007 ┆        5.4 ┆       797 │
│ tt0000008 ┆        5.4 ┆      2061 │
│ tt0000009 ┆        5.2 ┆       200 │
│ tt0000010 ┆        6.9 ┆      6949 │
│ tt0000011 ┆        5.3 ┆       356 │
│ …         ┆          … ┆         … │
└───────────┴────────────┴───────────┘
```

The translation problem

SQLite

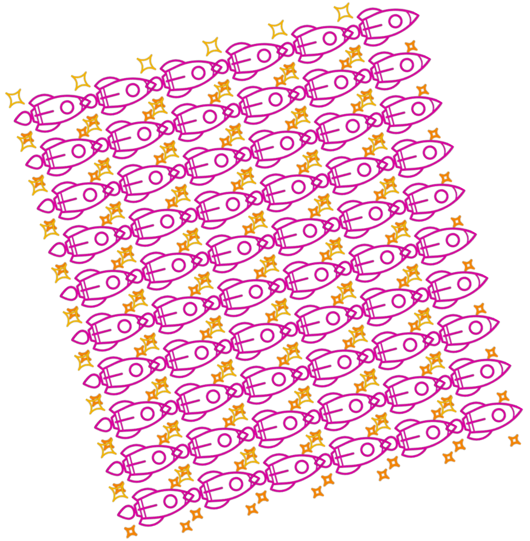
```
SELECT
  tconst,
  CAST(averageRating AS REAL(53)) as avg_rating,
  CAST(numVotes AS INTEGER) as num_votes
FROM ratings
```

PostgreSQL

```
SELECT
  tconst,
  CAST(averageRating AS DOUBLE PRECISION) as avg_rating,
  CAST(numVotes AS BIGINT) as num_votes
FROM ratings
```

The parameterization problem

One big query?



Or many small(er) queries?

“I want to write it in Python”

We like Python and we want to use it.

“I don’t want to write SQL”

SQL can be very concise

Some operations are hard to spell

Recursive common table expressions anyone?

What’s ~~the~~ a solution?

Presented with:

- Translation problem
- Parameterization problem
- Want to use Python
- Don’t want to write a bunch of SQL strings

Some people*, when presented with a problem, think, “I know, I’ll generate strings!”...

*Everyone at some point

Recall our simple example

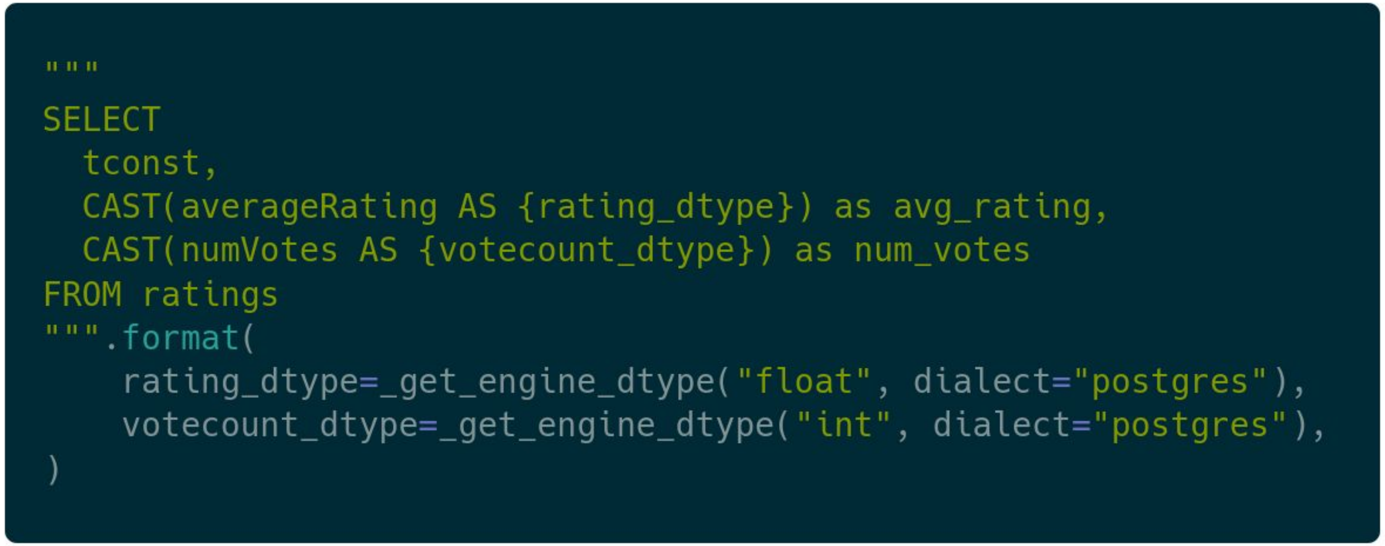
SQLite

```
SELECT
  tconst,
  CAST(averageRating AS REAL(53)) as avg_rating,
  CAST(numVotes AS INTEGER) as num_votes
FROM ratings
```

PostgreSQL

```
SELECT
  tconst,
  CAST(averageRating AS DOUBLE PRECISION) as avg_rating,
  CAST(numVotes AS BIGINT) as num_votes
FROM ratings
```

Sure, it starts off simple enough…



```
"""
SELECT
  tconst,
  CAST(averageRating AS {rating_dtype}) as avg_rating,
  CAST(numVotes AS {votecount_dtype}) as num_votes
FROM ratings
""".format(
    rating_dtype=_get_engine_dtype("float", dialect="postgres"),
    votecount_dtype=_get_engine_dtype("int", dialect="postgres"),
)
```

But remember…

The translation problem

Function names differ (or don’t exist!)

Function argument order differs

SQL engines have optimized versions of certain common functions

Output formats vary wildly

…

The parameterization problem

If the parameters were straightforward, the work would already be done.

Eventually some parameters end up dependent on other conditions…

“Outside factors”

“All I’m saying is that it would be great if we could…”

“We’ll fix it later”

“This is a high priority request from…”

And the query grows and grows…

This is fine

```
"""
SELECT
  tconst,
  CAST(averageRating AS {rating_dtype}) as avg_rating,
  CAST(numVotes AS {votecount_dtype}) as num_votes
FROM ratings
""".format(
    rating_dtype=_get_engine_dtype("float", dialect="postgres"),
    votecount_dtype=_get_engine_dtype("int", dialect="postgres"),
)
```

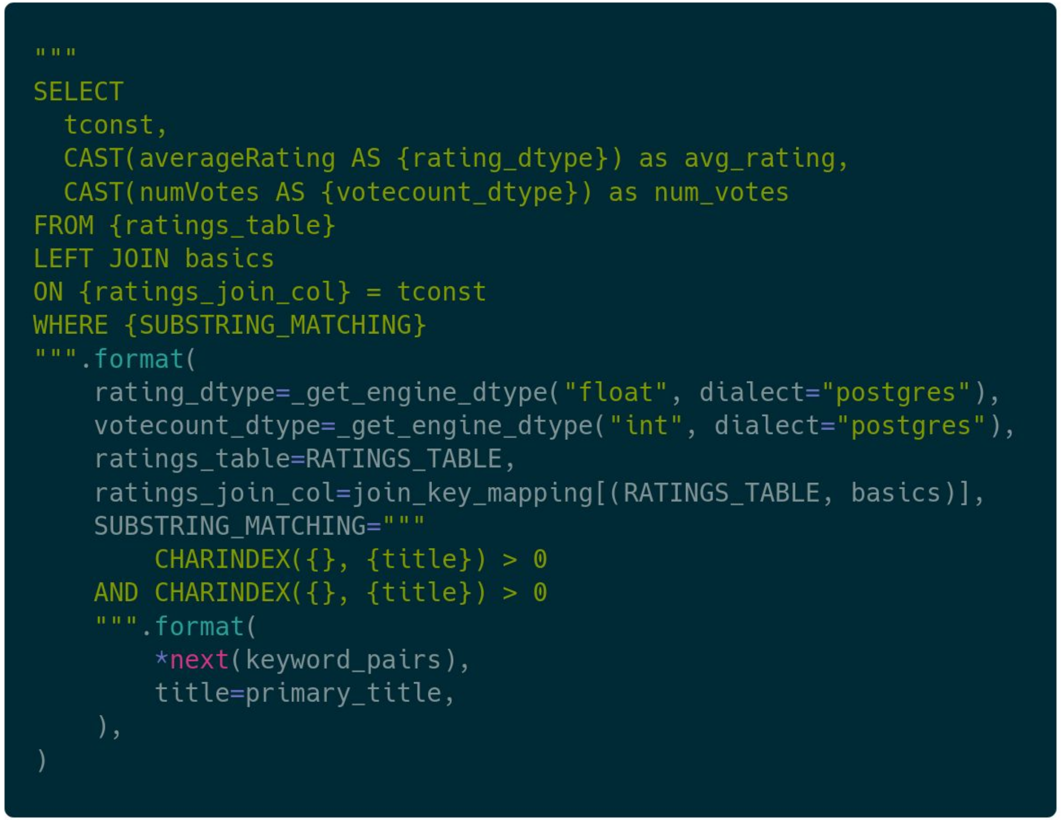
This is fine

```
"""
SELECT
  tconst,
  CAST(averageRating AS {rating_dtype}) as avg_rating,
  CAST(numVotes AS {votecount_dtype}) as num_votes
FROM ratings
LEFT JOIN basics
ON tconst
""".format(
    rating_dtype=_get_engine_dtype("float", dialect="postgres"),
    votecount_dtype=_get_engine_dtype("int", dialect="postgres"),
)
```

This is fine?

```
"""
SELECT
  tconst,
  CAST(averageRating AS {rating_dtype}) as avg_rating,
  CAST(numVotes AS {votecount_dtype}) as num_votes
FROM {ratings_table}
LEFT JOIN basics
ON {ratings_join_col} = tconst
""".format(
    rating_dtype=_get_engine_dtype("float", dialect="postgres"),
    votecount_dtype=_get_engine_dtype("int", dialect="postgres"),
    ratings_table=RATINGS_TABLE,
    ratings_join_col=join_key_mapping[(RATINGS_TABLE, basics)],
)
```

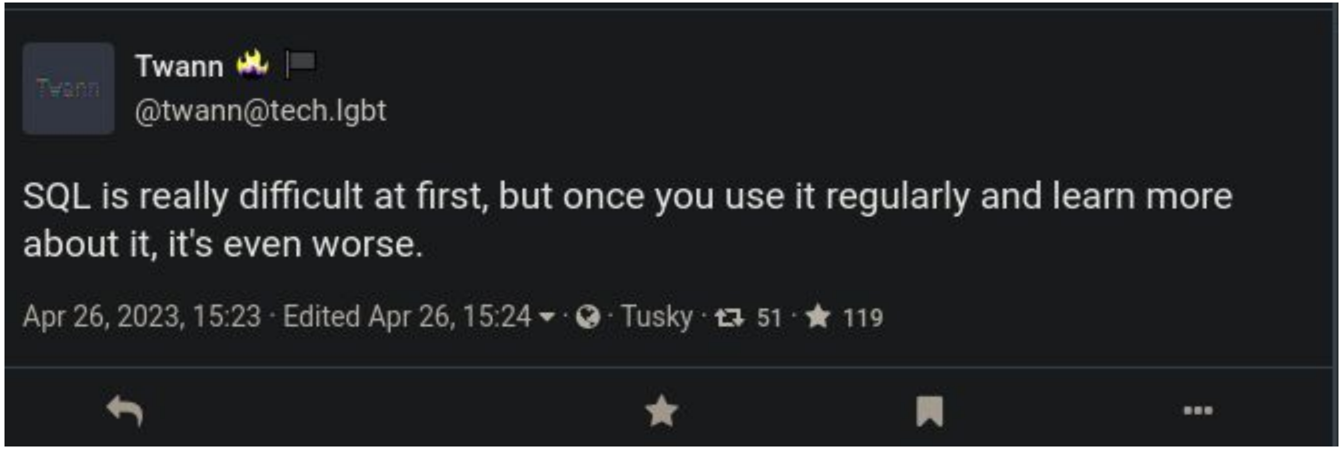
This ~~is fine.~~



```
"""
SELECT
  tconst,
  CAST(averageRating AS {rating_dtype}) as avg_rating,
  CAST(numVotes AS {votecount_dtype}) as num_votes
FROM {ratings_table}
LEFT JOIN basics
ON {ratings_join_col} = tconst
WHERE {SUBSTRING_MATCHING}
""".format(
    rating_dtype=_get_engine_dtype("float", dialect="postgres"),
    votecount_dtype=_get_engine_dtype("int", dialect="postgres"),
    ratings_table=RATINGS_TABLE,
    ratings_join_col=join_key_mapping[(RATINGS_TABLE, basics)],
    SUBSTRING_MATCHING="""
        CHARINDEX({}, {title}) > 0
    AND CHARINDEX({}, {title}) > 0
    """.format(
        *next(keyword_pairs),
        title=primary_title,
    ),
)
```

This is fine.

```
"""
SELECT
  tconst,
  CAST(averageRating AS {rating_dtype}) as avg_rating,
  CAST(numVotes AS {votecount_dtype}) as num_votes
FROM {ratings_table}
LEFT JOIN basics
ON {ratings_join_col} = tconst
WHERE {SUBSTRING_MATCHING}
""".format(
    rating_dtype=_get_engine_dtype("float", dialect="postgres"),
    votecount_dtype=_get_engine_dtype("int", dialect="postgres"),
    ratings_table=RATINGS_TABLE,
    ratings_join_col=join_key_mapping[(RATINGS_TABLE, basics)],
    SUBSTRING_MATCHING=(
        """
        CHARINDEX({}, {title}) > 0
    AND CHARINDEX({}, {title}) > 0
    """
        if ENGINE_SUPPORTS_CHARINDEX
        else """
        {title} LIKE '%{}%'
        AND {title} LIKE '%{}%'
        """
    ).format(
        *next(keyword_pairs),
        title=_get_engine_string_escape_fn("postgres")(primary_title)
    ),
)
```



Twann
@twann@tech.lgbt

SQL is really difficult at first, but once you use it regularly and learn more about it, it's even worse.

Apr 26, 2023, 15:23 · Edited Apr 26, 15:24 · · Tusky · 51 · 119

No thanks, I’m not going to use SQL

But remember, it’s *everywhere*.

Where does that leave us?

SQL standards are… not exactly standard*

SQL can be a little convoluted

String generation is madness

But we still want to write our analytics in Python

And we want to take advantage of modern query engines

What if…

instead of generating strings “by hand”, you use a type-safe DataFrame API that *eventually* generates strings?

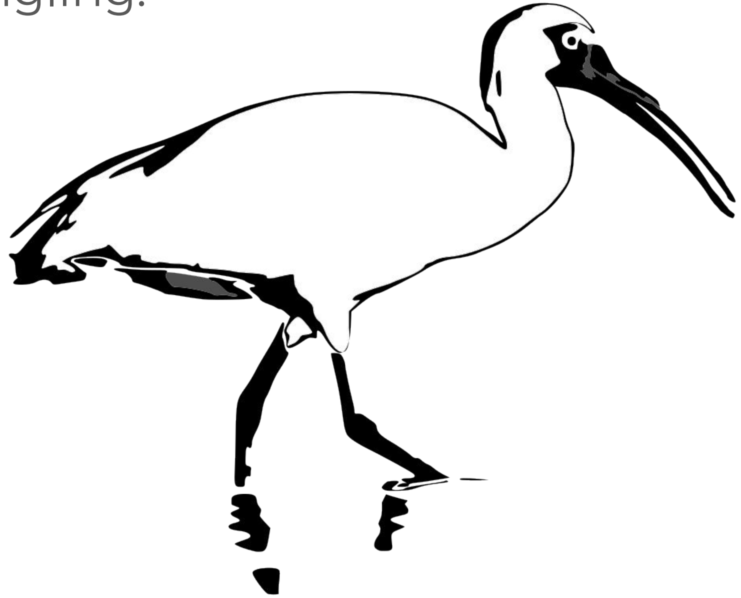
Ibis!

A lightweight Python library for data wrangling.

A dataframe API for Python

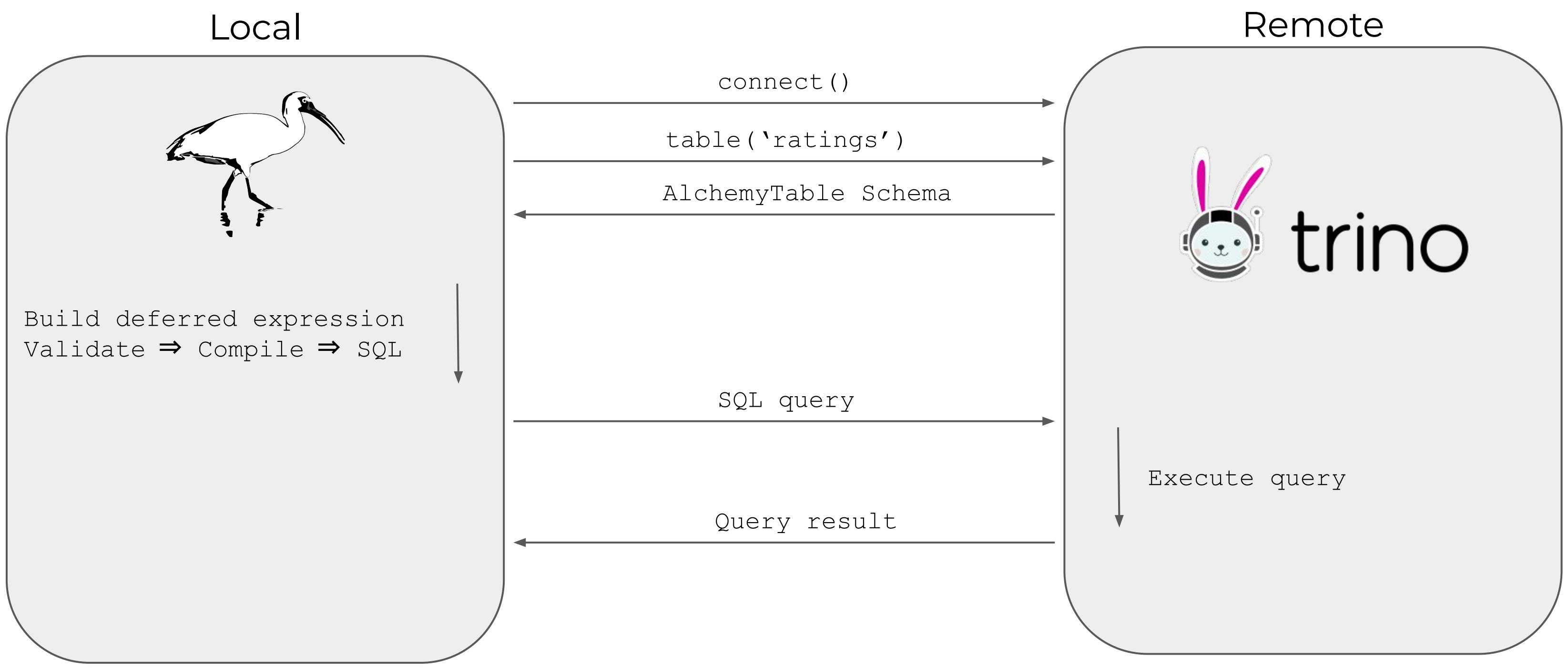
Interfaces to 16+ query engines

Deferred execution model

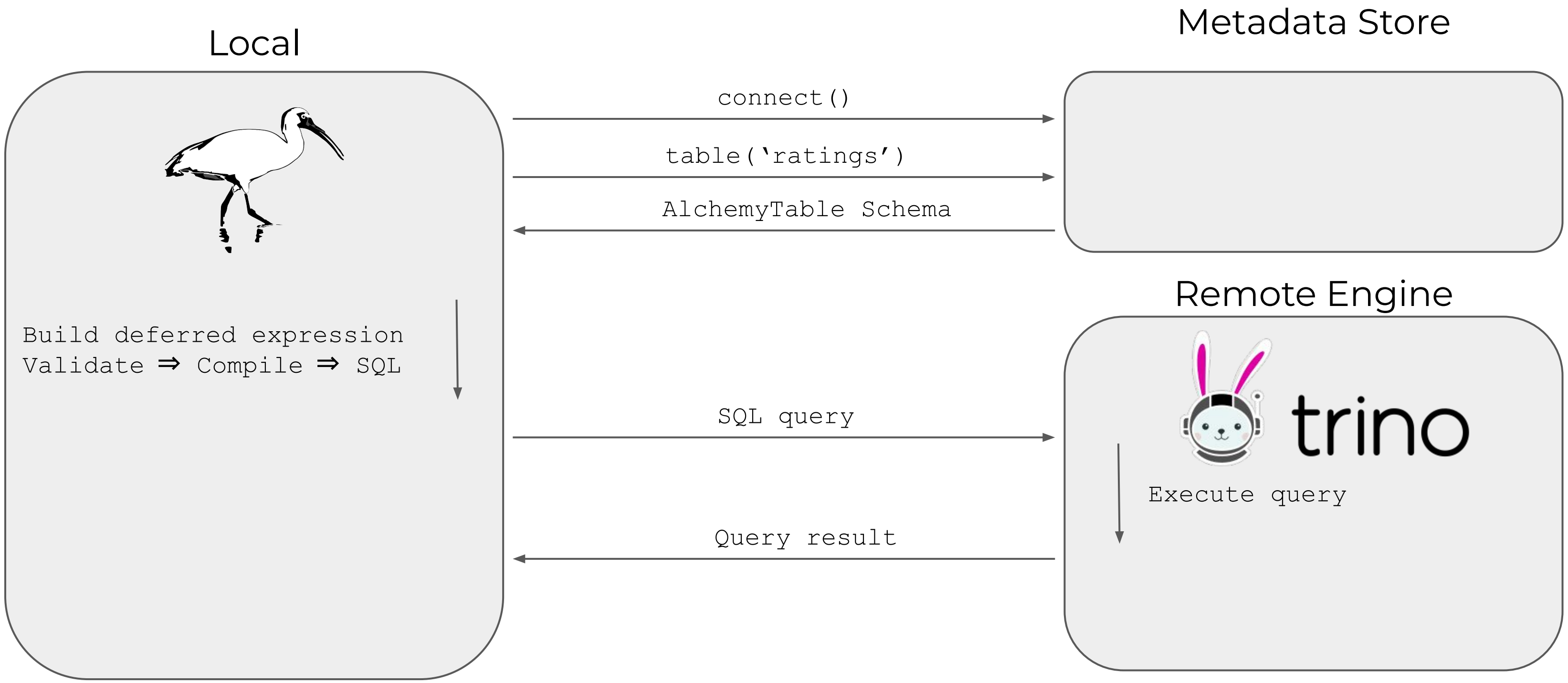


For any R-stats people in the room, it’s similar to dplyr / dbplyr

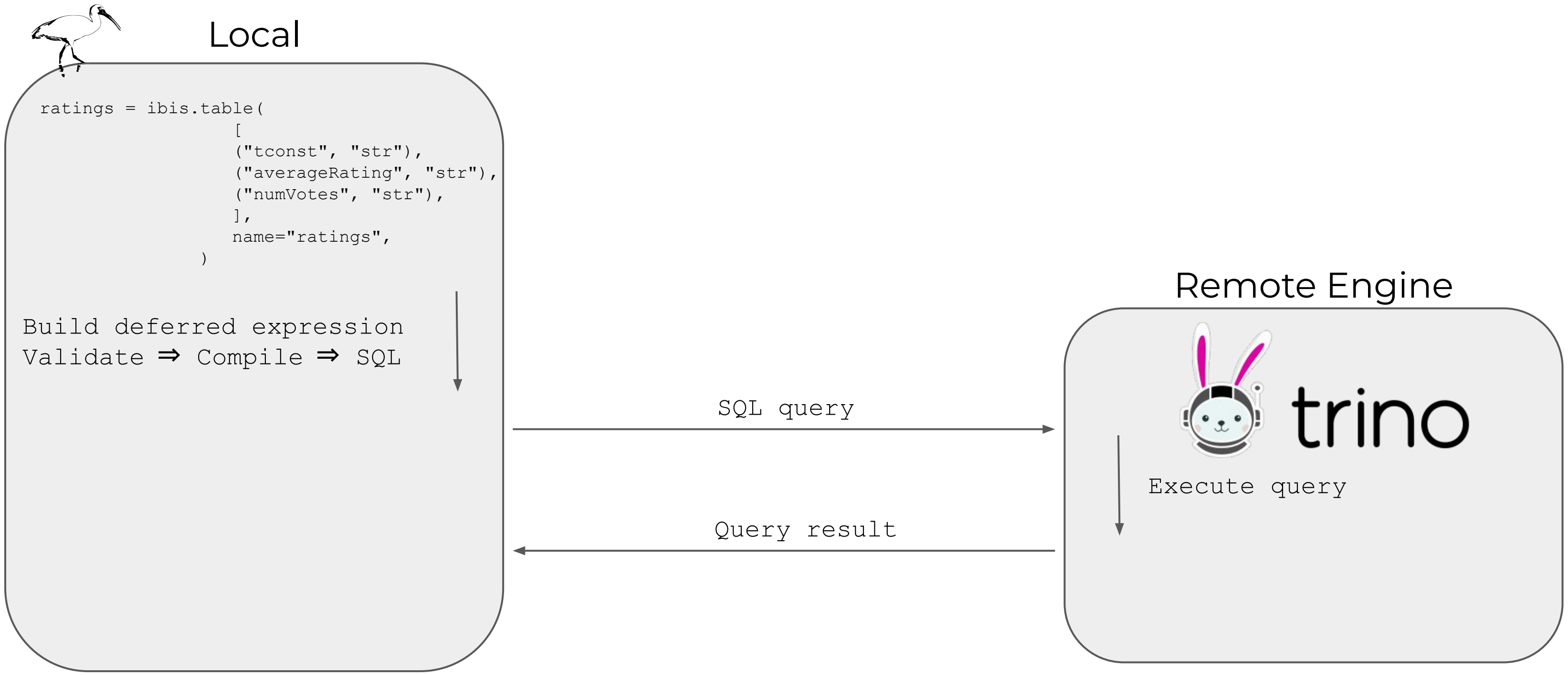
Deferred Execution



Deferred Execution



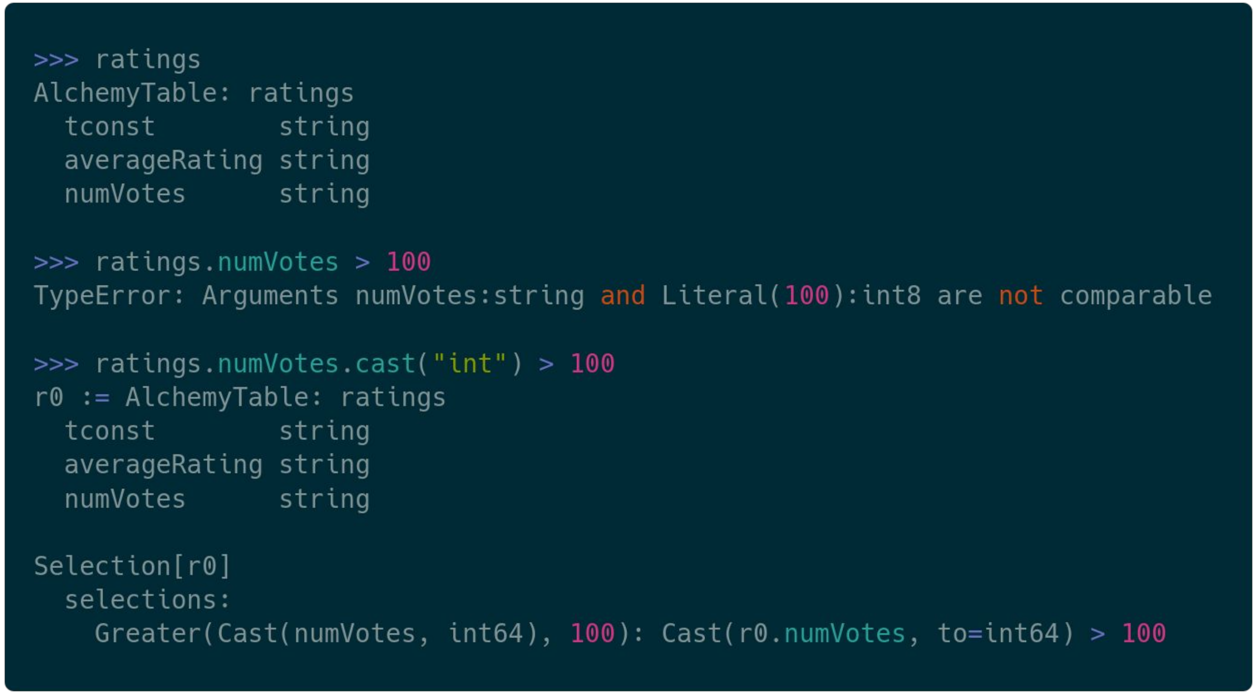
Deferred Execution



```
ratings = ibis.table(
                  [
                  ("tconst", "str"),
                  ("averageRating", "str"),
                  ("numVotes", "str"),
                  ],
                  name="ratings",
               )
```

Validation

Ibis validates expressions at construction – no execution required!



```
>>> ratings
AlchemyTable: ratings
  tconst        string
  averageRating string
  numVotes      string

>>> ratings.numVotes > 100
TypeError: Arguments numVotes:string and Literal(100):int8 are not comparable

>>> ratings.numVotes.cast("int") > 100
r0 := AlchemyTable: ratings
  tconst        string
  averageRating string
  numVotes      string

Selection[r0]
  selections:
    Greater(Cast(numVotes, int64), 100): Cast(r0.numVotes, to=int64) > 100
```

Demo Time!

Wait, what did I just see?

- select
- filter
- aggregate
- join
- Easily combine expressions

Supported backends

- ClickHouse
- BigQuery
- Dask
- DataFusion
- Druid
- DuckDB
- Impala
- mssql
- MySQL
- Oracle
- pandas
- Polars
- Postgres
- PySpark
- Snowflake
- SQLite
- Trino

Scale from dev to prod with less rewriting

BUT: There are no golden tickets

- Floating point math exists
- Regexen
- Data-dependent function behavior

It will definitely be less work than rewriting pandas as a spark DF

What’s next?

Cross-dialect `.sql()` support - 6.0

Unified UDF API - 6.0

More DDL support - 6.0

More (simpler) logical optimizations

New backends? - 6.0 (oracle)

Your Request Here

Questions?

https://ibis-project.org/

ibis-project/ibis

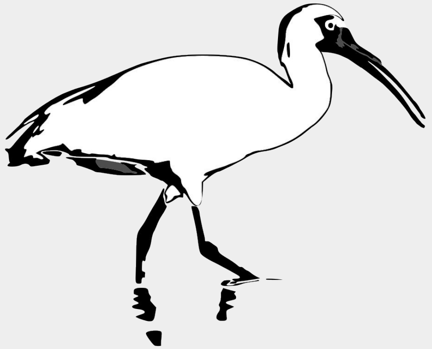
IbisData

ibis-dev/Lobby

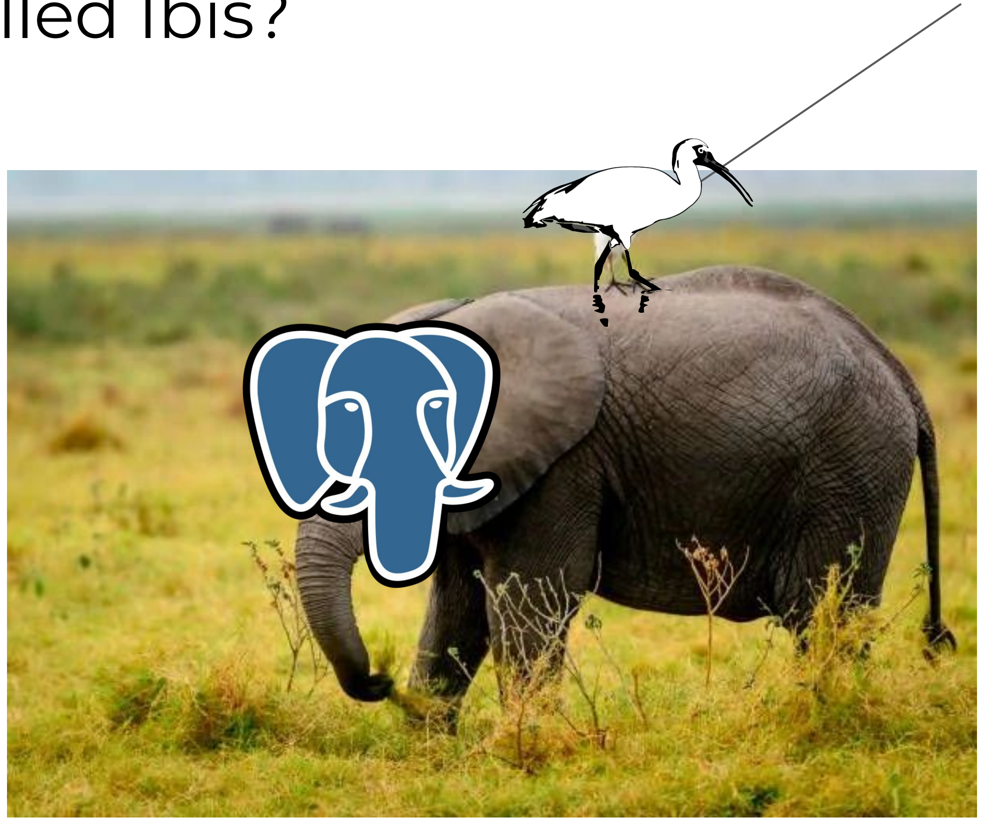
Phillip in the Cloud
cpcloud

```
pip install ibis-framework
pip install ibis-framework[trino]
pip install ibis-framework[$backend]

conda install -c conda-forge ibis-framework
                             ibis-bigquery
                             ibis-clickhouse
                             ibis-dask
                             ibis-datafusion
                             ibis-duckdb
                             ibis-impala
                             ibis-mysql
                             ibis-oracle
                             ibis-polars
                             ibis-postgres
                             ibis-pyspark
                             ibis-snowflake
                             ibis-sqlite
                             ibis-trino
```



Why is it called Ibis?



Can it read {parquet, csv, json, S3, etc...}?

Yes!

```
ibis.read_csv(“my_local.csv”)
ibis.read_csv(“my_local.csv.gz”)
ibis.read_parquet(“my_local.parquet”)
ibis.read_parquet(“path/to/folder/of/*.parquet”)
ibis.read_parquet(“s3://bucket/o/*.parquet”)
```

How does this compare to {PySpark, ...}?

The answer to, "how does Ibis compare to X?" is "Ibis helps you use X."

I have a big dataframe in memory already, can I use Ibis with it?

You bet.

```
ibis.memtable(some_big_df, name=”cool_new_table”)
```