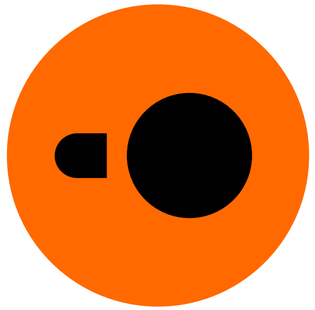


DuckDB Spatial: Supercharged Geospatial SQL

Max Gabrielsson
Software Engineer

DuckDB Labs

About me

Max Gabrielsson

- BsC from Uppsala University
- Fan turned Soft. Engineer @ DuckDB Labs
- Execution layer & Spatial Extension

DuckDB Labs

- Founded by DuckDB creators
- Based in Amsterdam, ≈18 people
- Development & Support for DuckDB

DuckDB Labs

What is DuckDB?

- Analytical embedded SQL Database
- Originally research project @ CWI
- Free and open source (MIT)
- No external dependencies
- Currently in pre-release (v0.10.3)
- Extensive python/client integrations
- Makes the most of your laptop

```
$ pip install duckdb
```

In-process

Portable

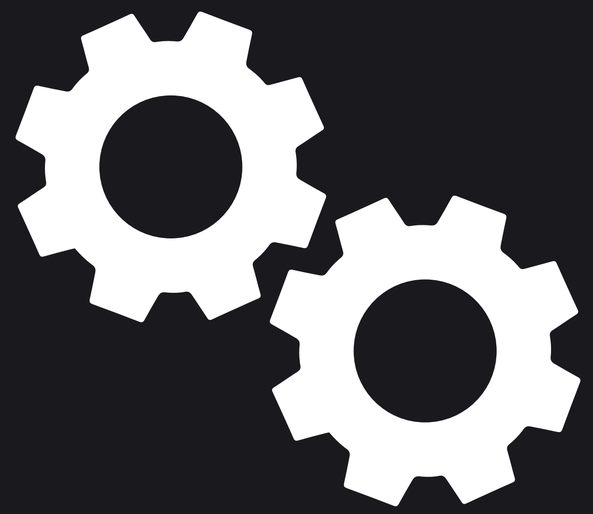
Fast

Open-source

The three sides of DuckDB

Execution **Storage** **UX**

Execution



Query Processing

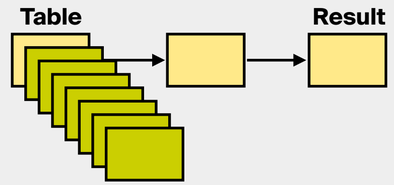
Row-at-a-Time

- Classic Database approach
- Low memory footprint
- High CPU overhead

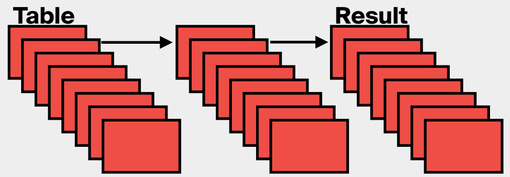
Column-at-a-Time

- Common in Dataframes
- Efficient CPU usage - SIMD
- Large memory footprint

Row-at-a-Time



Column-at-a-Time

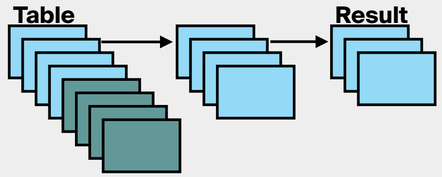


Query Processing

DuckDB is *Vector-at-a-Time*

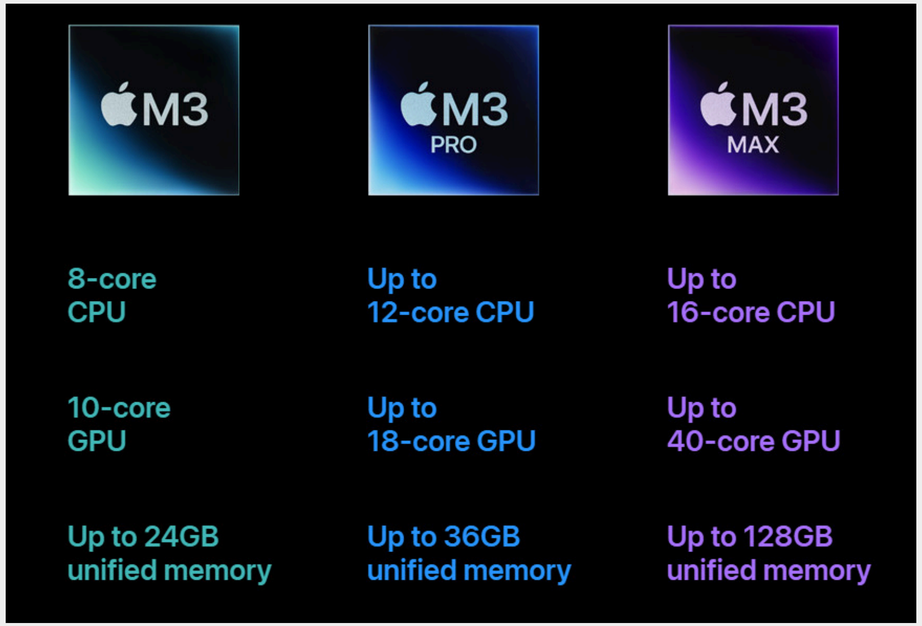
- Best of both worlds!
- Optimize for CPU-Cache

Vectorized Processing

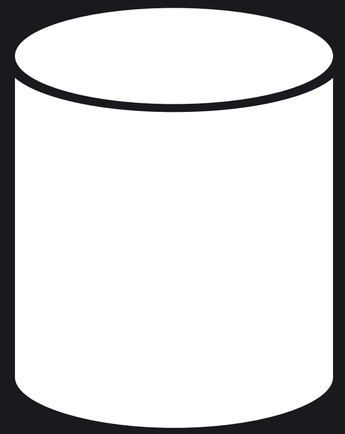


DuckDB is *Multi-threaded*

- Natural to parallelize over vectors
- Parallelism is increasingly important!
- Optionally order-preserving



Storage



Every ~~analytics~~ GIS project ever

You need a database!!

DuckDB's Storage

- DuckDB is not an in-memory DB
- 1 Database = 1 File
- Updates, "ACID", Transactions
- Multiple tables in one database
- Table columns stored individually
  - Fast search w/ per-column statistics
  - Compression: save 3-5x on storage!
  - Paradoxically improves query speed

Compression Example: TPC-H (SF1)

DuckDB Version	Size	Ratio	Compression	Date
0.2.8	0.85 GB	1.00×	None	2021-07
0.2.9	0.79 GB	1.07×	RLE + Constant	2021-09
0.3.2	0.56 GB	1.51×	Bitpacking	2022-01
0.3.3	0.32 GB	2.64×	Dictionary	2022-04
0.5.0	0.29 GB	2.93×	Frame of reference	2022-09
0.6.0	0.17 GB	5.00×	FSST + CHIMP	2022-11

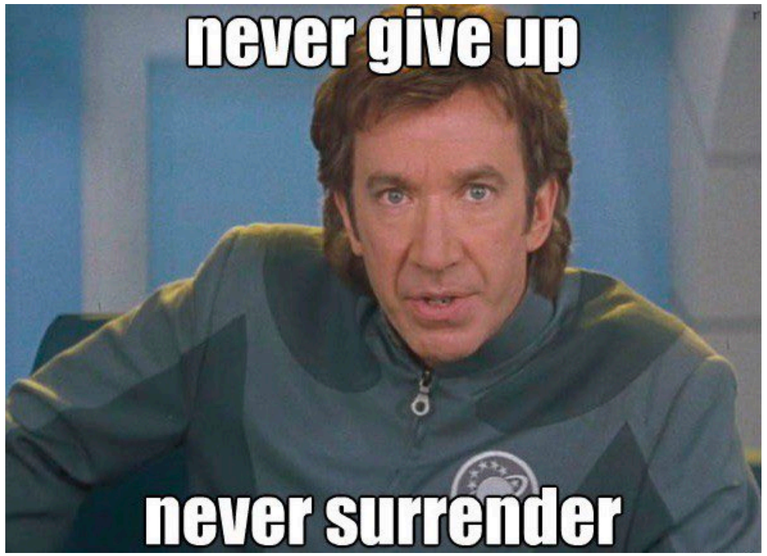
UX

In Process Deployment

- No separate server
- No config, no env, no docker
- Minimal transfer delay/overhead
- Run DuckDB as part of larger system
  - In the backend
  - In the client
  - In the browser (WASM)
  - In your phone

“Graceful Degradation”

- What to do when out of RAM?
- Almost all operators "spill to disk”
- “Never crash, always make progress”

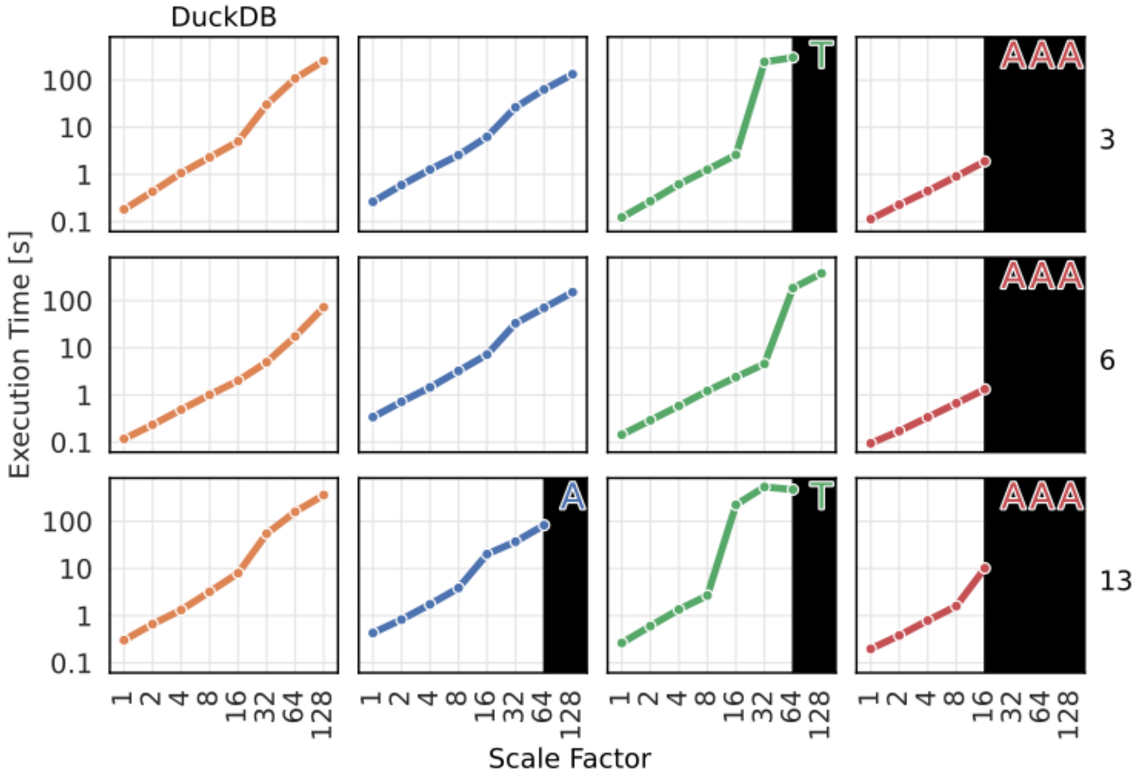


Robust External Hash Aggregation in the Solid State Age

Laurens Kuiper
CWI, Amsterdam, Netherlands
laurens.kuiper@cwi.nl

Peter Boncz
CWI, Amsterdam, Netherlands
peter.boncz@cwi.nl

Hannes Mühleisen
CWI, Amsterdam, Netherlands
hannes.muehleisen@cwi.nl



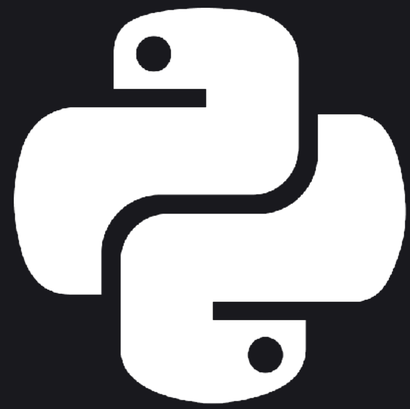
“Friendly SQL”

- Superset of PostgreSQL dialect
- FROM-first syntax, trailing commas
- GROUP BY ALL, column selection
- Nested types, lambda functions
- Python inspired

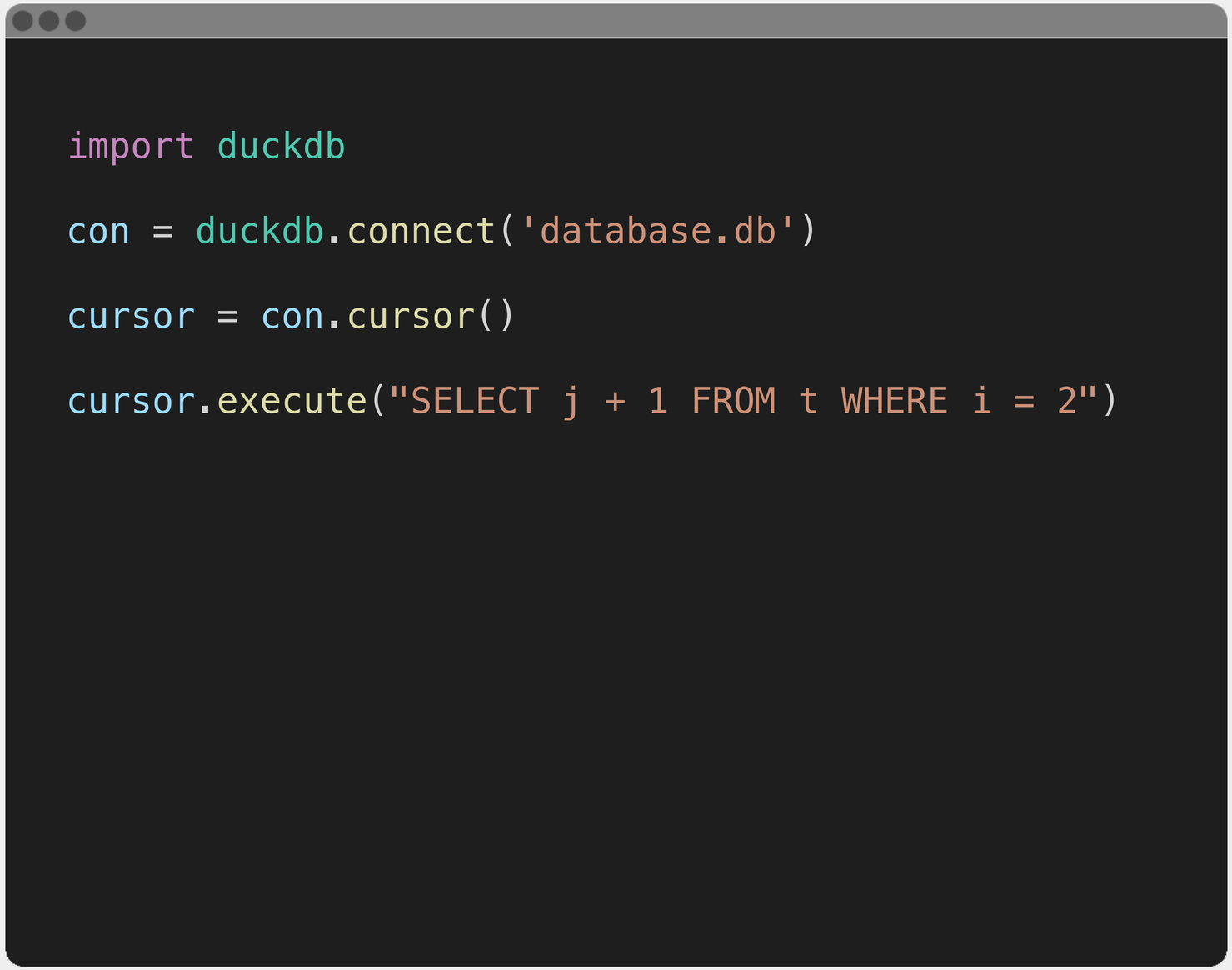
Speaking of python...

```sql
-- FROM-first syntax!
FROM my_table SELECT 1 + 2;

-- Trailing commas
SELECT a, b, c, FROM my_table;

-- Column selection
SELECT * EXCLUDE (id, name) FROM users

-- List comprehensions
SELECT [x + 1 for x in [1, 2, 3]];

- Unified function call syntax
SELECT name.upper() FROM users
```

DuckDB & Python

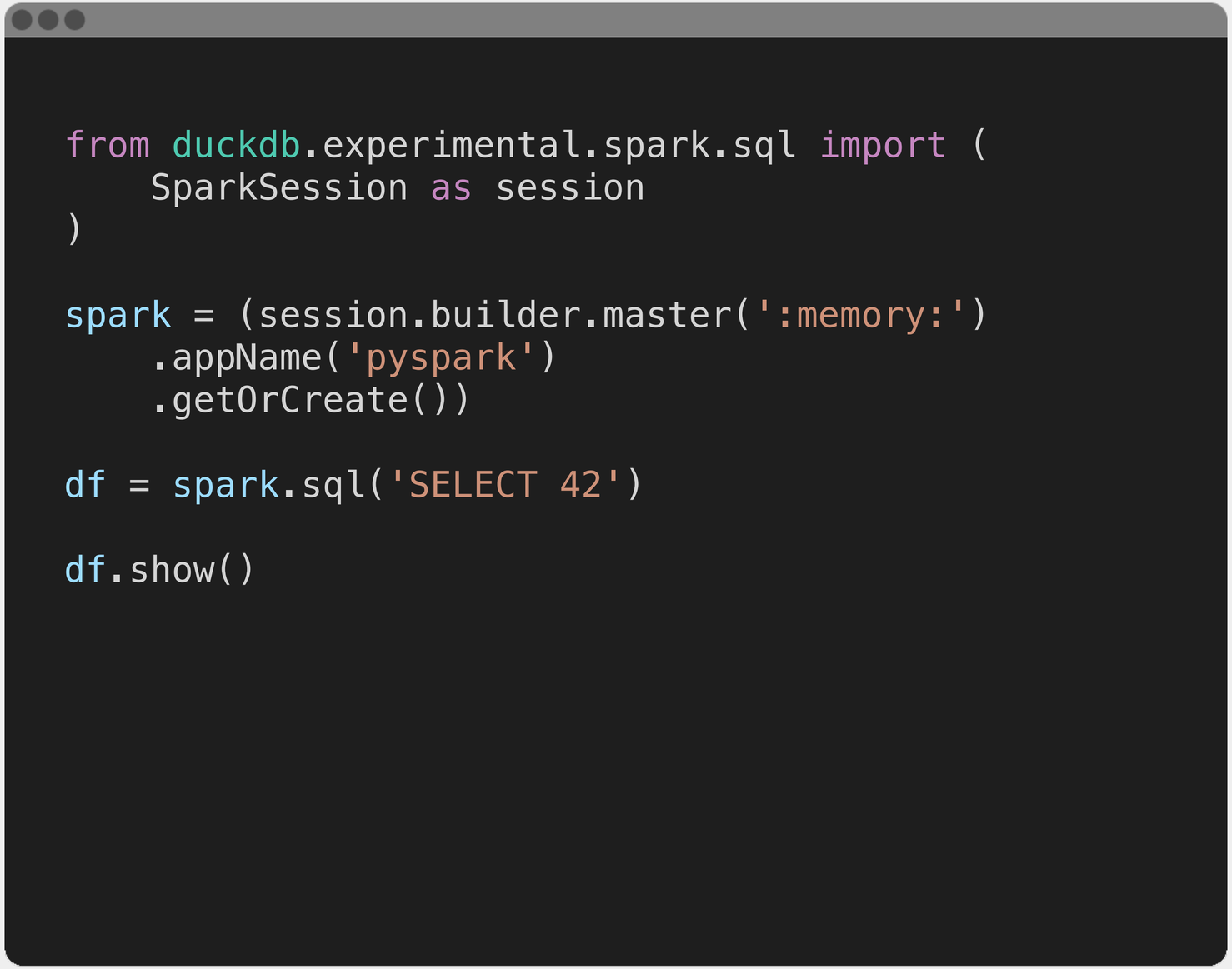


Python DB API 2.0



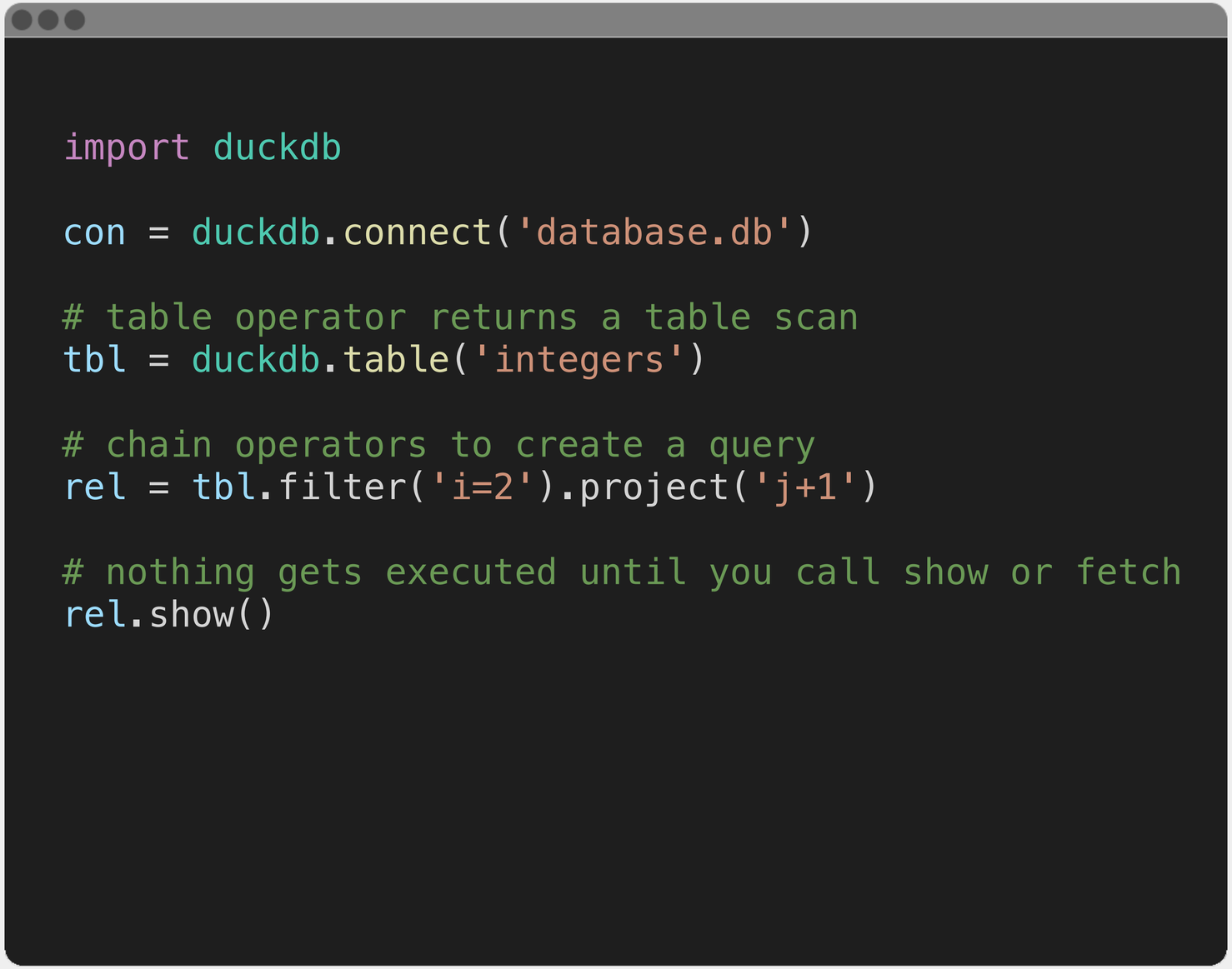
```python
import duckdb

con = duckdb.connect('database.db')

cursor = con.cursor()

cursor.execute("SELECT j + 1 FROM t WHERE i = 2")
```

PySpark API (Experimental)



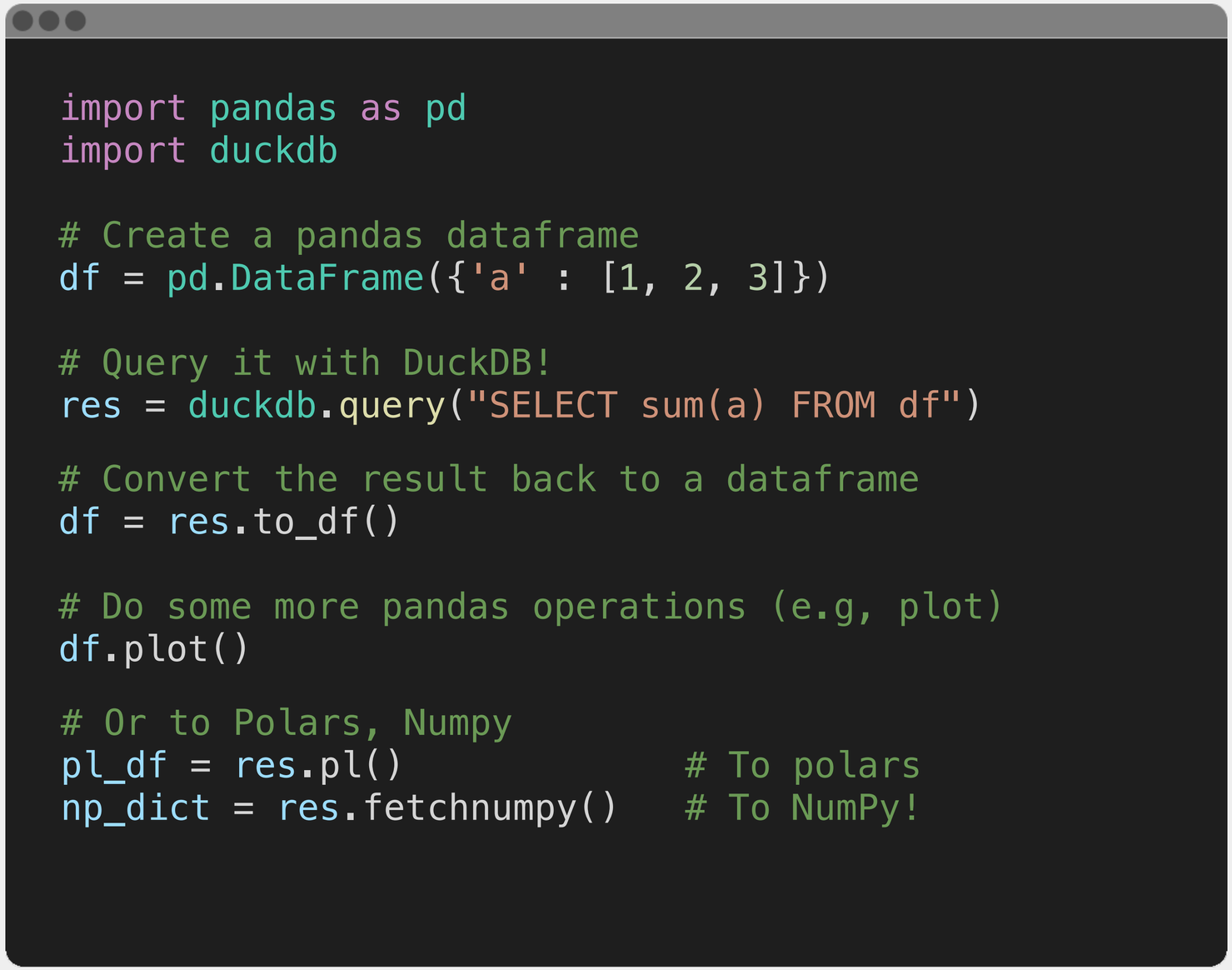
```python
from duckdb.experimental.spark.sql import (
    SparkSession as session
)

spark = (session.builder.master(':memory:')
    .appName('pyspark')
    .getOrCreate())

df = spark.sql('SELECT 42')

df.show()
```

Relational API



```python
import duckdb

con = duckdb.connect('database.db')

# table operator returns a table scan
tbl = duckdb.table('integers')

# chain operators to create a query
rel = tbl.filter('i=2').project('j+1')

# nothing gets executed until you call show or fetch
rel.show()
```

“Zero-Copy” Dataframe interoperability



```
import pandas as pd
import duckdb

# Create a pandas dataframe
df = pd.DataFrame({'a' : [1, 2, 3]})

# Query it with DuckDB!
res = duckdb.query("SELECT sum(a) FROM df")

# Convert the result back to a dataframe
df = res.to_df()

# Do some more pandas operations (e.g, plot)
df.plot()

# Or to Polars, Numpy
pl_df = res.pl()                # To polars
np_dict = res.fetchnumpy()      # To NumPy!
```

“Zero-Copy” Dataframe interoperability

- Scan dataframes and arrays in scope, directly from memory!
- Polars, Numpy, Pandas, PyArrow
- Product of in-process deployment
- Use the best tool for the job!

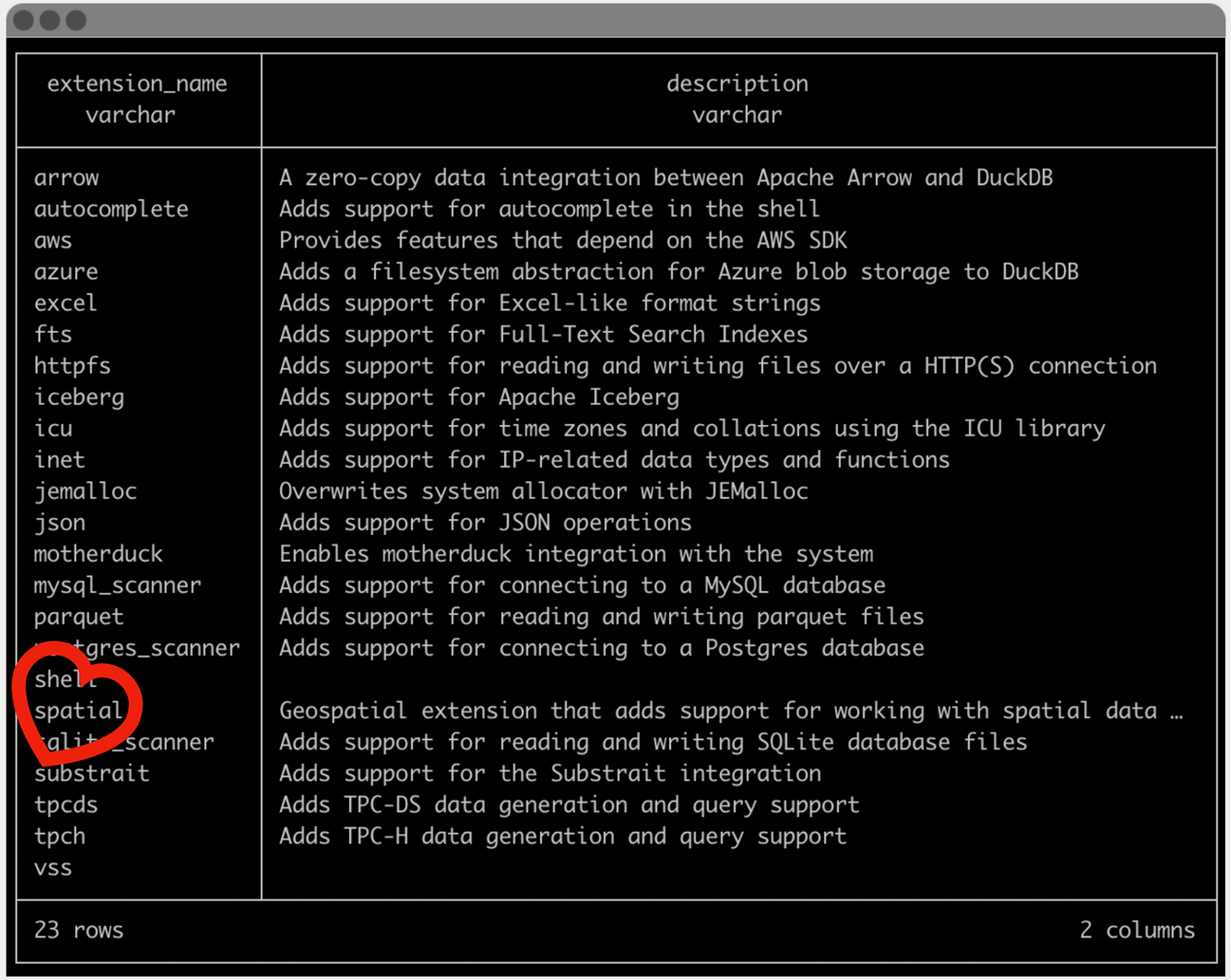
```python
import pandas as pd
import duckdb

# Create a pandas dataframe
df = pd.DataFrame({'a' : [1, 2, 3]})

# Query it with DuckDB!
res = duckdb.query("SELECT sum(a) FROM df")

# Convert the result back to a dataframe
df = res.to_df()

# Do some more pandas operations (e.g, plot)
df.plot()

# Or to Polars, Numpy
pl_df = res.pl()                # To polars
np_dict = res.fetchnumpy()      # To NumPy!
```

Ok, what’s missing?

DuckDB Spatial

DuckDB Extensions

- Compiled code modules
- (Down)loadable at runtime
- Types, functions, operators, scans
- You can write your own!
- Anything non-essential
- JSON/Postgres/MySQL/SQLite
- And of course...

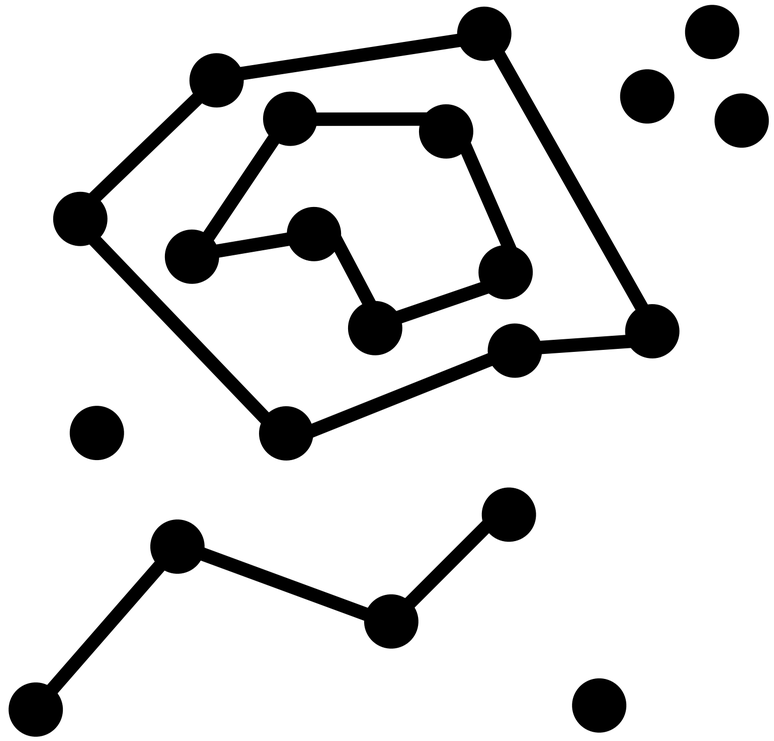


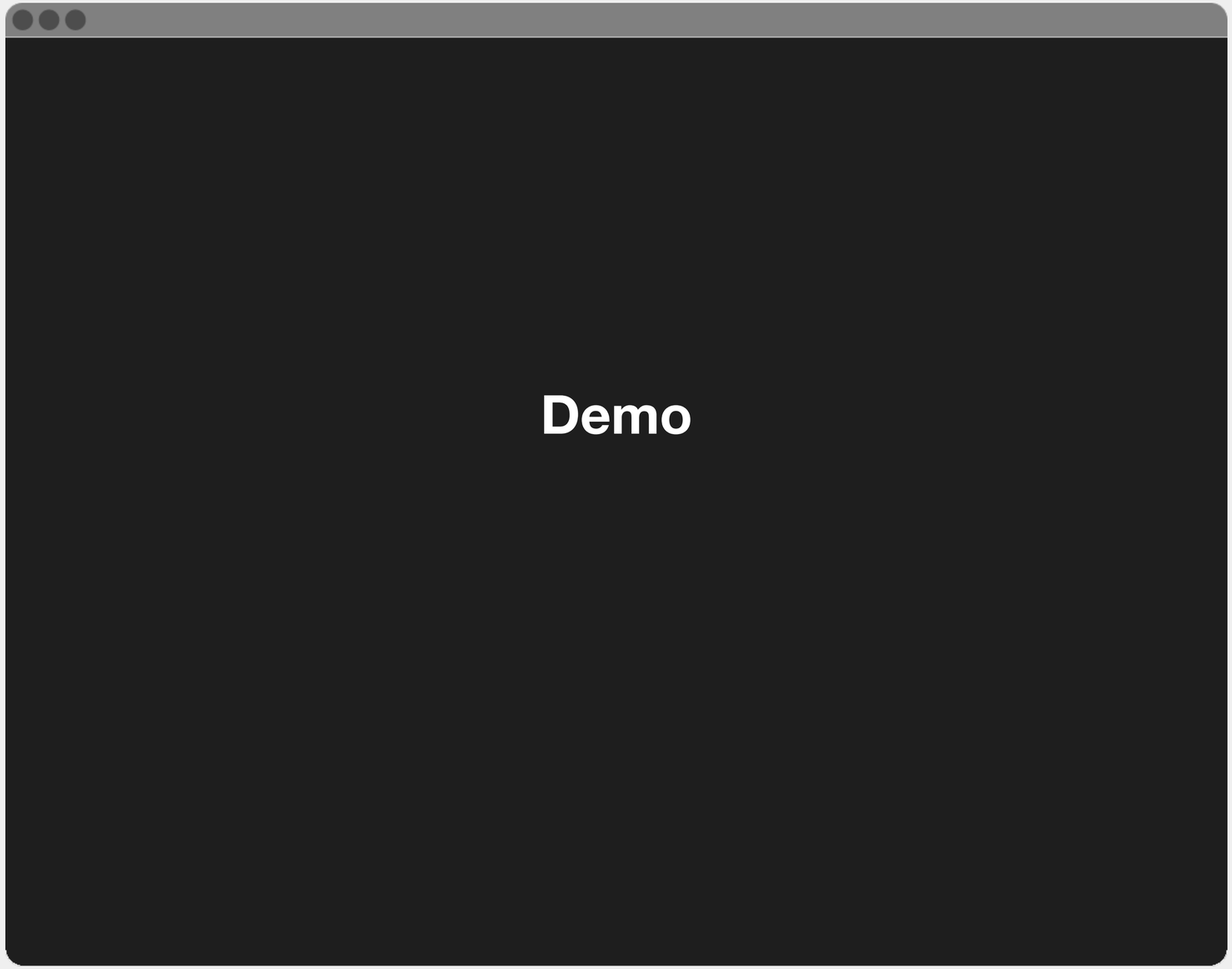
extension_name varchar	description varchar
arrow	A zero-copy data integration between Apache Arrow and DuckDB
autocomplete	Adds support for autocomplete in the shell
aws	Provides features that depend on the AWS SDK
azure	Adds a filesystem abstraction for Azure blob storage to DuckDB
excel	Adds support for Excel-like format strings
fts	Adds support for Full-Text Search Indexes
httpfs	Adds support for reading and writing files over a HTTP(S) connection
iceberg	Adds support for Apache Iceberg
icu	Adds support for time zones and collations using the ICU library
inet	Adds support for IP-related data types and functions
jemalloc	Overwrites system allocator with JEMalloc
json	Adds support for JSON operations
motherduck	Enables motherduck integration with the system
mysql_scanner	Adds support for connecting to a MySQL database
parquet	Adds support for reading and writing parquet files
postgres_scanner	Adds support for connecting to a Postgres database
shell	
spatial	Geospatial extension that adds support for working with spatial data …
sqlite_scanner	Adds support for reading and writing SQLite database files
substrait	Adds support for the Substrait integration
tpcds	Adds TPC-DS data generation and query support
tpch	Adds TPC-H data generation and query support
vss	

23 rows 2 columns

DuckDB Spatial

- Officially supported extension adding geospatial capabilities
- “Simple Features” vector **`GEOMETRY`** type
- Modeled after PostGIS, 100+ “ST_” functions
- No runtime dependencies
  - Statically bundles GDAL/GEOS/PROJ
  - Embedded projection database
  - Pre-built binaries for 10 platforms




Demo

```
# Import DuckDB
import duckdb


# Install and load the spatial extension
duckdb.execute('INSTALL spatial')
duckdb.execute('LOAD spatial')



# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)
```

```
# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)


# Show the schema of the “rel” relation
duckdb.sql('SUMMARIZE rel').select('column_name', 'column_type').show()
```

```
# Show the schema of the “rel” relation
duckdb.sql('SUMMARIZE rel').select('column_name', 'column_type').show()
```

column_name	column_type
varchar	varchar
vendor_id	VARCHAR
pickup_datetime	VARCHAR
dropoff_datetime	VARCHAR
passenger_count	BIGINT
trip_distance	DOUBLE
pickup_longitude	DOUBLE
pickup_latitude	DOUBLE
rate_code	VARCHAR
store_and_fwd_flag	VARCHAR
dropoff_longitude	DOUBLE
dropoff_latitude	DOUBLE
payment_type	VARCHAR
fare_amount	DOUBLE
surcharge	DOUBLE
mta_tax	DOUBLE
tip_amount	DOUBLE
tolls_amount	DOUBLE
total_amount	DOUBLE
18 rows	2 columns

```
# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)


# Show the schema of the “rel” relation
duckdb.sql('SUMMARIZE rel').select('column_name', 'column_type').show()
```

```
# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)


# Create a 'rides' table with geometry columns for the pickup and dropoff points
# along with the reported trip distance
rel.select("""
    st_point(pickup_latitude, pickup_longitude) as pickup_point,
    st_point(dropoff_latitude, dropoff_longitude) as dropoff_point,
    trip_distance,
""").to_table('rides')
```

```
# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)


# Create a 'rides' table with geometry columns for the pickup and dropoff points
# transformed into ESRI:102718 CRS, along with the reported trip distance
rel.select("""
    st_transform(
        st_point(pickup_latitude, pickup_longitude),
            'EPSG:4326',
            'ESRI:102718'
        ) as pickup_point,
    st_transform(
        st_point(dropoff_latitude, dropoff_longitude),
           'EPSG:4326',
           'ESRI:102718'
        ) as dropoff_point,
    trip_distance,
""").to_table('rides')
```

```
expr = 'st_point(pickup_latitude, pickup_longitude)'
```

```
expr = duckdb.FunctionExpression('st_point', ‘pickup_latitude’, ‘pickup_longitude')
```

```
def make_point(x_col: duckdb.Expression, y_col: duckdb.Expression):
    return duckdb.FunctionExpression('st_point', x_col, y_col)

def transform(col: duckdb.Expression, source_crs: str, target_crs: str):
    source_expr = duckdb.ConstantExpression(source_crs)
    target_expr = duckdb.ConstantExpression(target_crs)
    return duckdb.FunctionExpression('st_transform', col, source_expr, target_expr)


# Attach the transform method to the Expression class
duckdb.Expression.transform = transform
```

```
# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)

def make_point(x_col: duckdb.Expression, y_col: duckdb.Expression):
def transform(col: duckdb.Expression, source_crs: str, target_crs: str):
duckdb.Expression.transform = transform


# Create a 'rides' table with geometry columns for the pickup and dropoff points
# transformed into ESRI:102718 CRS, along with the reported trip distance
rel.select("""
    st_transform(
        st_point(pickup_latitude, pickup_longitude),
            'EPSG:4326',
            'ESRI:102718'
        ) as pickup_point,
    st_transform(
        st_point(dropoff_latitude, dropoff_longitude),
           'EPSG:4326',
           'ESRI:102718'
        ) as dropoff_point,
    trip_distance,
""").to_table('rides')
```

```
# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)

def make_point(x_col: duckdb.Expression, y_col: duckdb.Expression):
def transform(col: duckdb.Expression, source_crs: str, target_crs: str):
duckdb.Expression.transform = transform


# Create a 'rides' table with geometry columns for the pickup and dropoff points
# transformed into ESRI:102718 CRS, along with the reported trip distance
rel.select(
    make_point('pickup_latitude', 'pickup_longitude')
        .transform(
            'EPSG:4326',
            'ESRI:102718'
        ).alias('pickup_point'),
    make_point('dropoff_latitude', 'dropoff_longitude')
        .transform(
            'EPSG:4326',
            'ESRI:102718'
        ).alias('dropoff_point'),
    'trip_distance',
).to_table('rides')
```

```
# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)

# Create a 'rides' table with geometry columns for the pickup and dropoff points
# transformed into ESRI:102718 CRS, along with the reported trip distance
rel.select(
    make_point('pickup_latitude', 'pickup_longitude')
        .transform(
            'EPSG:4326',
            'ESRI:102718'
        ).alias('pickup_point'),
    make_point('dropoff_latitude', 'dropoff_longitude')
        .transform(
            'EPSG:4326',
            'ESRI:102718'
        ).alias('dropoff_point'),
    'trip_distance',
).to_table('rides')
```

```
# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)

# Create a 'rides' table with geometry columns for the pickup and dropoff points
# transformed into ESRI:102718 CRS, along with the reported trip distance
rides = rel.select(
    make_point('pickup_latitude', 'pickup_longitude')
        .transform(
            'EPSG:4326',
            'ESRI:102718'
        ).alias('pickup_point'),
    make_point('dropoff_latitude', 'dropoff_longitude')
        .transform(
            'EPSG:4326',
            'ESRI:102718'
        ).alias('dropoff_point'),
    'trip_distance',
    (distance('pickup_point', 'dropoff_point') / 5280).alias('aerial_distance')
)


# How many rides have a distance discrepancy?
rides.filter('trip_distance < aerial_distance').count('*')
```

```
# How many rides have a distance discrepancy?
rides.filter('trip_distance < aerial_distance').count('*')
```

```
┌──────────────┐
│ count_star() │
│    int64     │
├──────────────┤
│        65104 │
└──────────────┘
```

```
# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)

# Create a 'rides' table with geometry columns for the pickup and dropoff points
# transformed into ESRI:102718 CRS, along with the reported trip distance
rides = rel.select(
    make_point('pickup_latitude', 'pickup_longitude')
        .transform(
            'EPSG:4326',
            'ESRI:102718'
        ).alias('pickup_point'),
    make_point('dropoff_latitude', 'dropoff_longitude')
        .transform(
            'EPSG:4326',
            'ESRI:102718'
        ).alias('dropoff_point'),
    'trip_distance',
    (distance('pickup_point', 'dropoff_point') / 5280).alias('aerial_distance')
)


# How many rides have a distance discrepancy?
rides.filter('trip_distance < aerial_distance').count('*')
```

```
# Load 1 million taxi rides from parquet
path = './nyc_taxi/yellow_tripdata_2010-01-limit1mil.parquet'
rel = duckdb.read_parquet(path)

# Create a 'rides' table with geometry columns for the pickup and dropoff points
# transformed into ESRI:102718 CRS, along with the reported trip distance
rel.select(
    make_point('pickup_latitude', 'pickup_longitude')
        .transform(
            'EPSG:4326',
            'ESRI:102718'
        ).alias('pickup_point'),
    make_point('dropoff_latitude', 'dropoff_longitude')
        .transform(
            'EPSG:4326',
            'ESRI:102718'
        ).alias('dropoff_point'),
    'trip_distance',
    (distance('pickup_point', 'dropoff_point') / 5280).alias('aerial_distance')
).filter('trip_distance > aerial_distance').to_table('cleaned_rides')
```

```
# Before we go further, list the available GDAL drivers
duckdb.table_function("st_drivers").show()
```

```
# Before we go further, list the available GDAL drivers
duckdb.table_function("st_drivers").show()
```

short_name varchar	long_name varchar	can_create boolean	can_copy boolean	can_open boolean	help_url varchar
ESRI Shapefile	ESRI Shapefile	true	false	true	https://gdal.org/drivers/vector/shapefi…
MapInfo File	MapInfo File	true	false	true	https://gdal.org/drivers/vector/mitab.h…
UK .NTF	UK .NTF	false	false	true	https://gdal.org/drivers/vector/ntf.html
LVBAG	Kadaster LV BAG Ex…	false	false	true	https://gdal.org/drivers/vector/lvbag.h…
S57	IHO S-57 (ENC)	true	false	true	https://gdal.org/drivers/vector/s57.html
·	·	·	·	·	·
·	·	·	·	·	·
·	·	·	·	·	·
PMTiles	ProtoMap Tiles	true	false	true	https://gdal.org/drivers/vector/pmtiles…
JSONFG	OGC Features and G…	true	false	true	https://gdal.org/drivers/vector/jsonfg.…
TIGER	U.S. Census TIGER/…	false	false	true	https://gdal.org/drivers/vector/tiger.h…
AVCBin	Arc/Info Binary Co…	false	false	true	https://gdal.org/drivers/vector/avcbin.…
AVCE00	Arc/Info E00 (ASCI…	false	false	true	https://gdal.org/drivers/vector/avce00.…

53 rows (10 shown) 6 columns

```
# Create a relation for scanning the taxi zones shapefile
zones = duckdb.sql("FROM st_read('./taxi_zones.shp')").set_alias('zones')

# Join the cleaned rides table with the taxi zones relation using the ST_WITHIN predicate
joined = duckdb.table('cleaned_rides').join(zones, 'st_within(dropoff_point, zones.geom)')


# Aggregate the number of trips per zone
zone_trips = joined.aggregate('zone, count(*) as trips', 'zone').order('trips DESC')

zone_trips.show()
```

```
zone_trips.show()
```

zone varchar	trips int64
East Village	33720
Times Sq/Theatre District	33462
Penn Station/Madison Sq West	30505
Midtown Center	27944
Clinton East	27654
Lincoln Square East	27438
Murray Hill	27241
Upper East Side North	26360
Upper East Side South	23800
Upper West Side South	23360
.	.
.	.
.	.
Oakwood	6
Rossville/Woodrow	6
Willets Point	6
Mariners Harbor	6
Eltingville/Annadale/Prince's Bay	5
Governor's Island/Ellis Island/Liberty Island	4
Charleston/Tottenville	4
Arden Heights	2

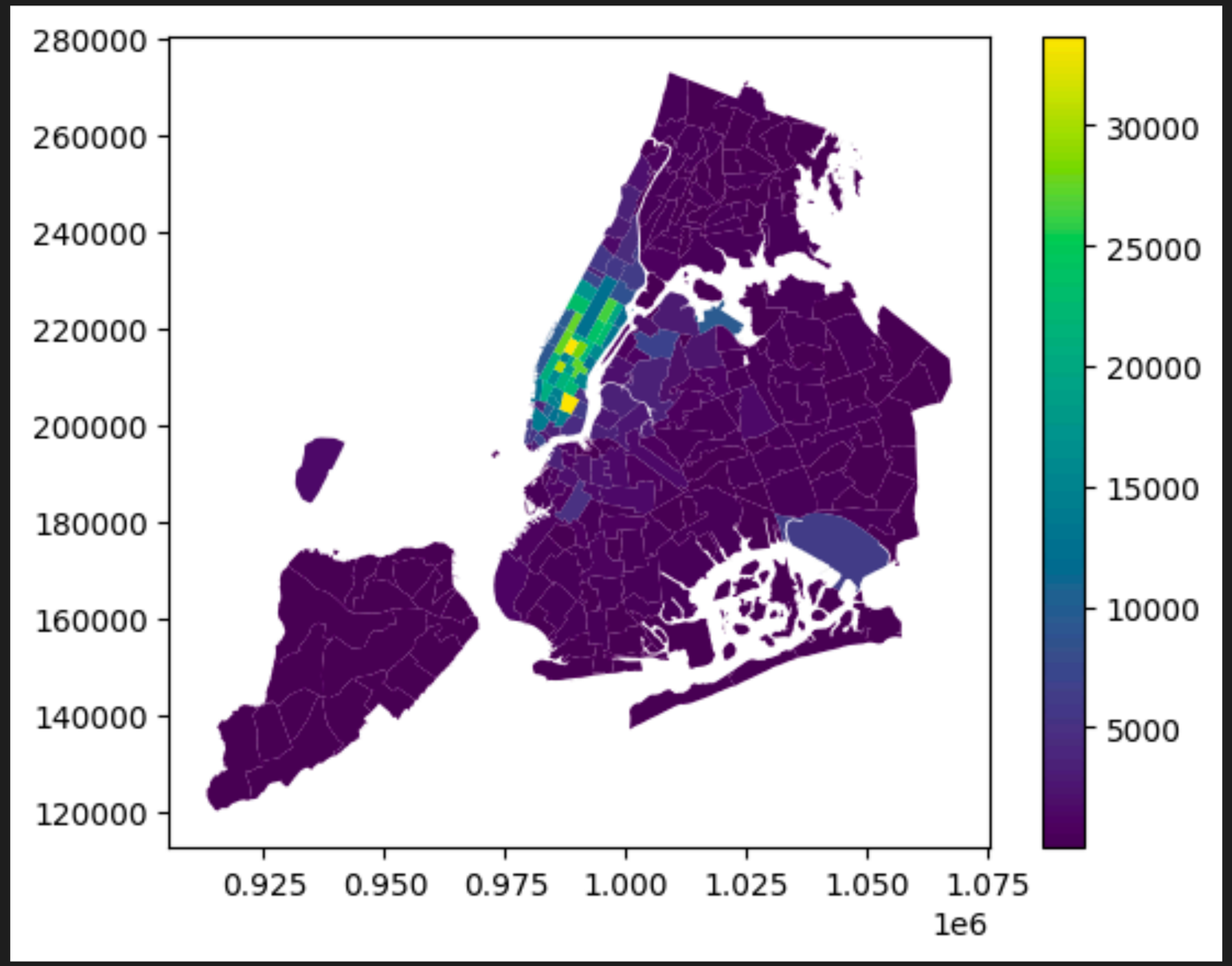
...

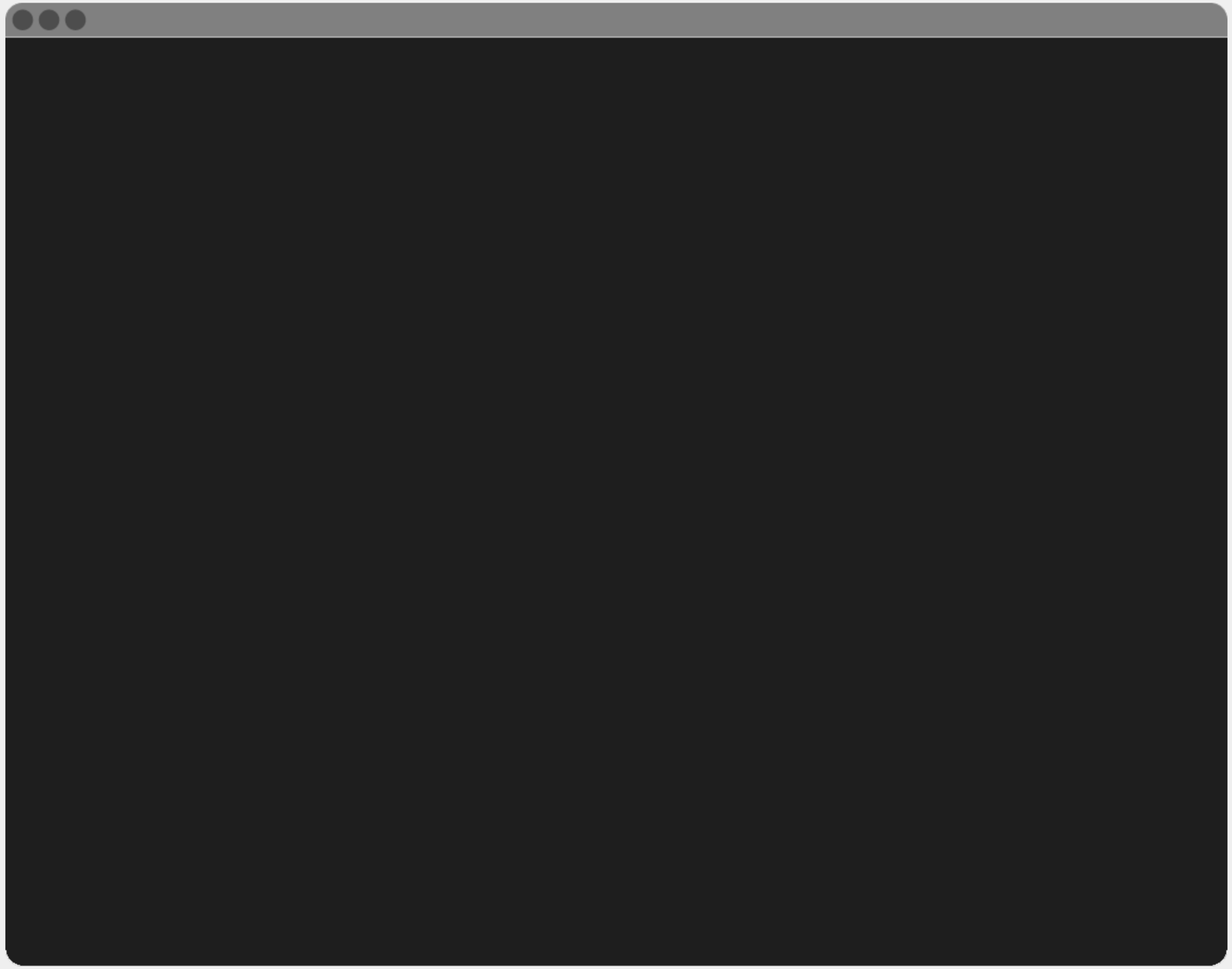
259 rows (20 shown) 2 columns

```
# Create a relation for scanning the taxi zones shapefile
zones = duckdb.sql("FROM st_read('./taxi_zones.shp')").set_alias('zones')

# Join the cleaned rides table with the taxi zones relation using the ST_WITHIN predicate
joined = duckdb.table('cleaned_rides').join(zones, 'st_within(dropoff_point, zones.geom)')


# Aggregate the number of trips per zone
zone_trips = joined.aggregate('zone, count(*) as trips', 'zone').order('trips DESC')


zone_trips.show()
```

```
# Create a relation for scanning the taxi zones shapefile
zones = duckdb.sql("FROM st_read('./taxi_zones.shp')").set_alias('zones')

# Join the cleaned rides table with the taxi zones relation using the ST_WITHIN predicate
joined = duckdb.table('cleaned_rides').join(zones, 'st_within(dropoff_point, zones.geom)')


# Aggregate the number of trips per zone, convert the zone geometry to WKT
zone_trips = joined.aggregate('zone, count(*) as trips, ST_AsText(any_value(geom)) as wkt'
                              , 'zone')

# Create a DataFrame from the result
df = zone_trips.to_df()


import geopandas as gpd
from matplotlib import pyplot as plt


# Create a GeoDataFrame from the DataFrame, converting the WKT column to a geometry column
gdf = gpd.GeoDataFrame(df, geometry=gpd.GeoSeries.from_wkt(df['wkt']), crs="ESRI:102718")

# Plot the number zones colored by the number of trips using GeoPandas
gdf.plot(column='trips', legend=True)
```

```
# Plot the number zones colored by the number of trips using GeoPandas
gdf.plot(column='trips', legend=True)
```



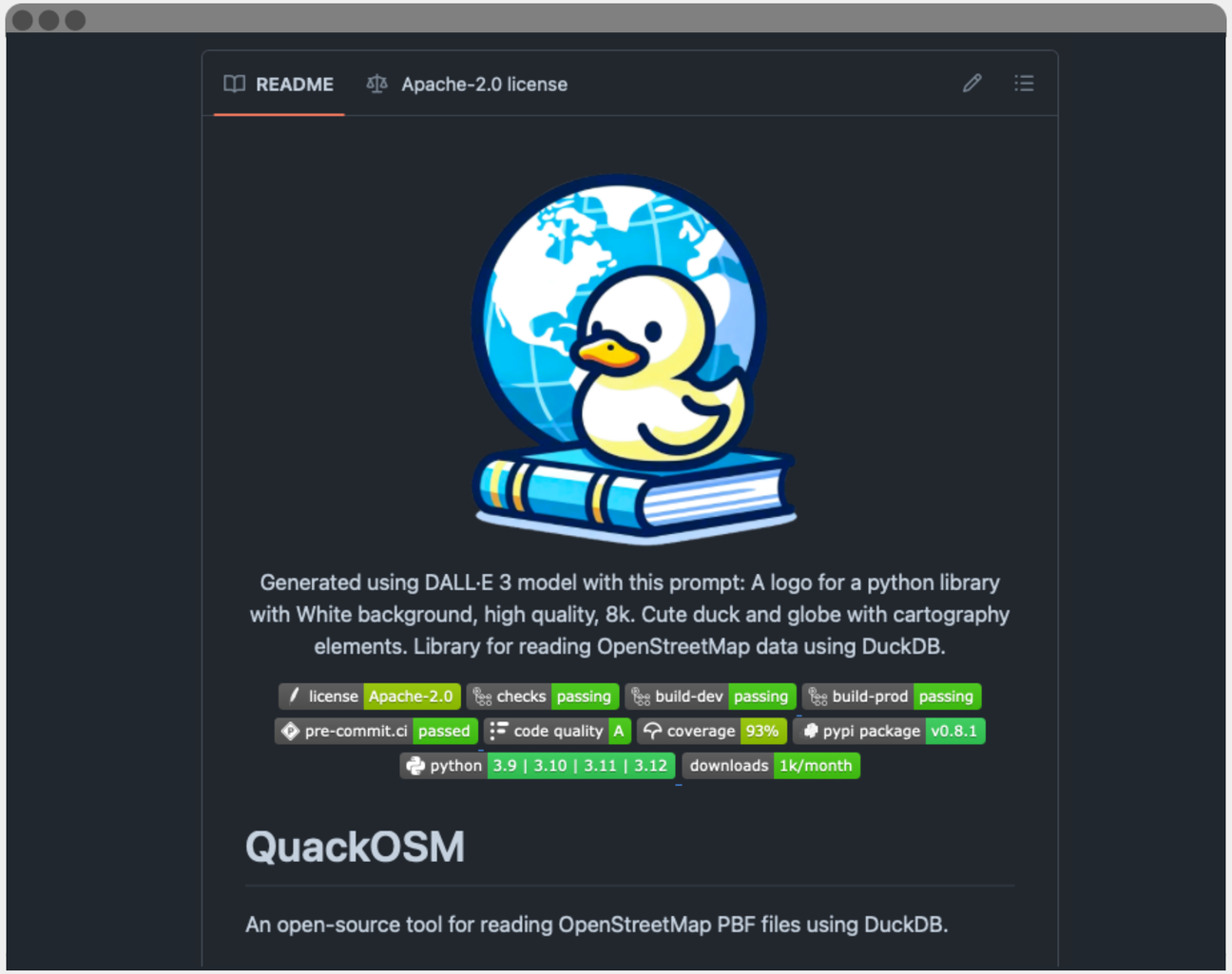


DuckDB Spatial Limitations & Future work

- So far focused on internal infrastructure
  - Geometry representation
  - Extensibility from within DuckDB
- Build on top of that: spatial indexes, projection handling, functions
- Documentation overhaul
- More work on integrations
  - GeoParquet, GeoArrow
  - Better support in client libraries

QuackOSM

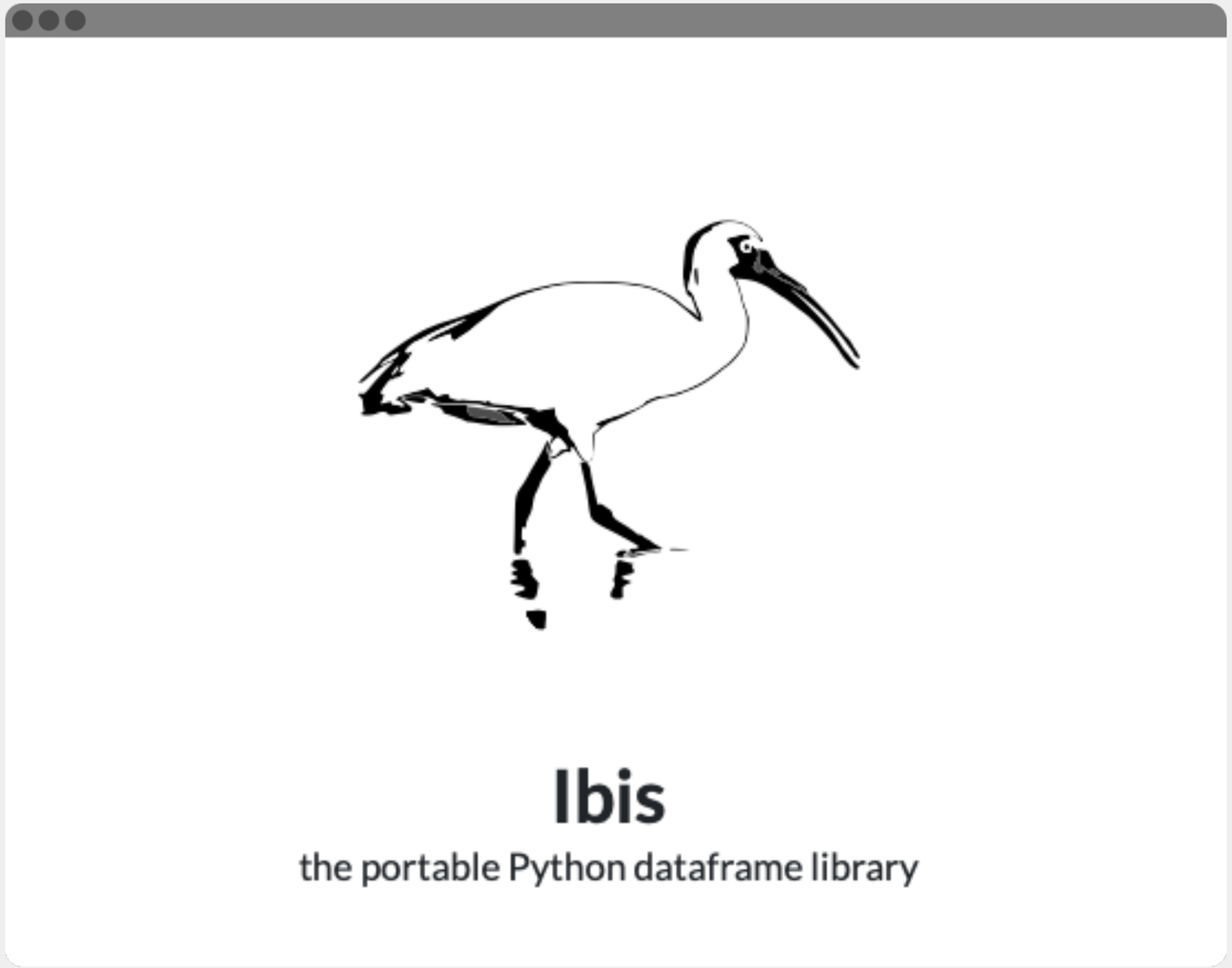
- Import OpenStreetMap exports
- Very convenient to filter clip/tags
- Automatic download/selection
- Export as GeoParquet
- Both CLI and Python module



https://github.com/kraina-ai/quackosm

The IBIS Project

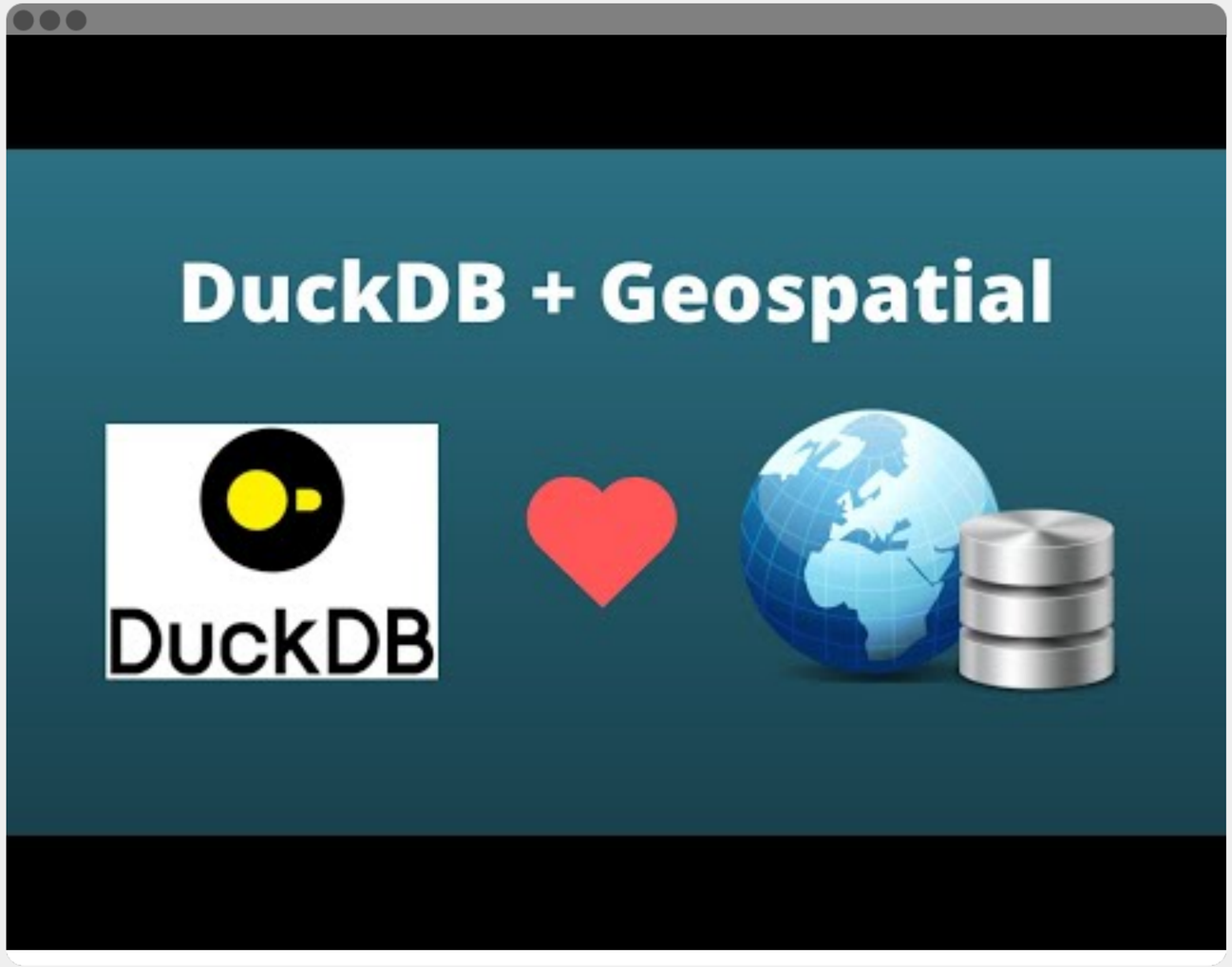
- Universal interface to DB's and DF's
- Great support for DuckDB
- Most common geospatial exprs
  - And GDAL import!



https://ibis-project.org/posts/ibis-duckdb-geospatial/

Dr. Qiusheng Wu's Spatial Data Management

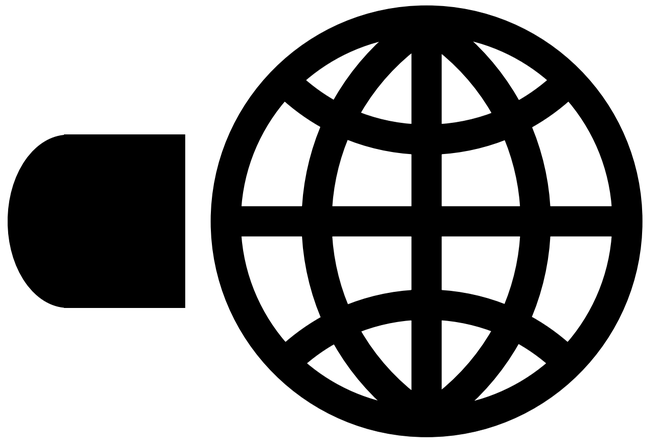
- University of Tennessee GEOG-414
- Course material available online
- Web-book + YouTube videos
- Python, GeeMap, DuckDB, PostGIS



https://geog-414.gishub.org/

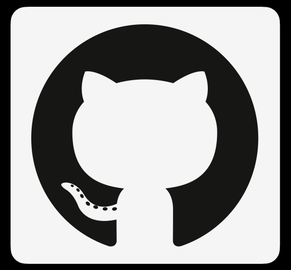
Conclusion

- DuckDB is a portable analytical embedded SQL database
- DuckDB best of databases and data frames
- Quickstart your next GIS workflow with the Spatial extension
- Lots of cool stuff on the way!



Stay in touch

discord.duckdb.org

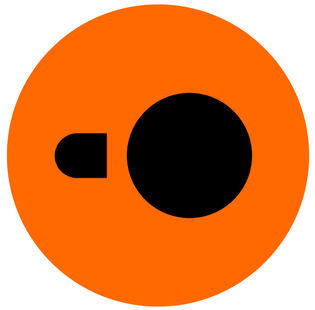


duckdb/duckdb

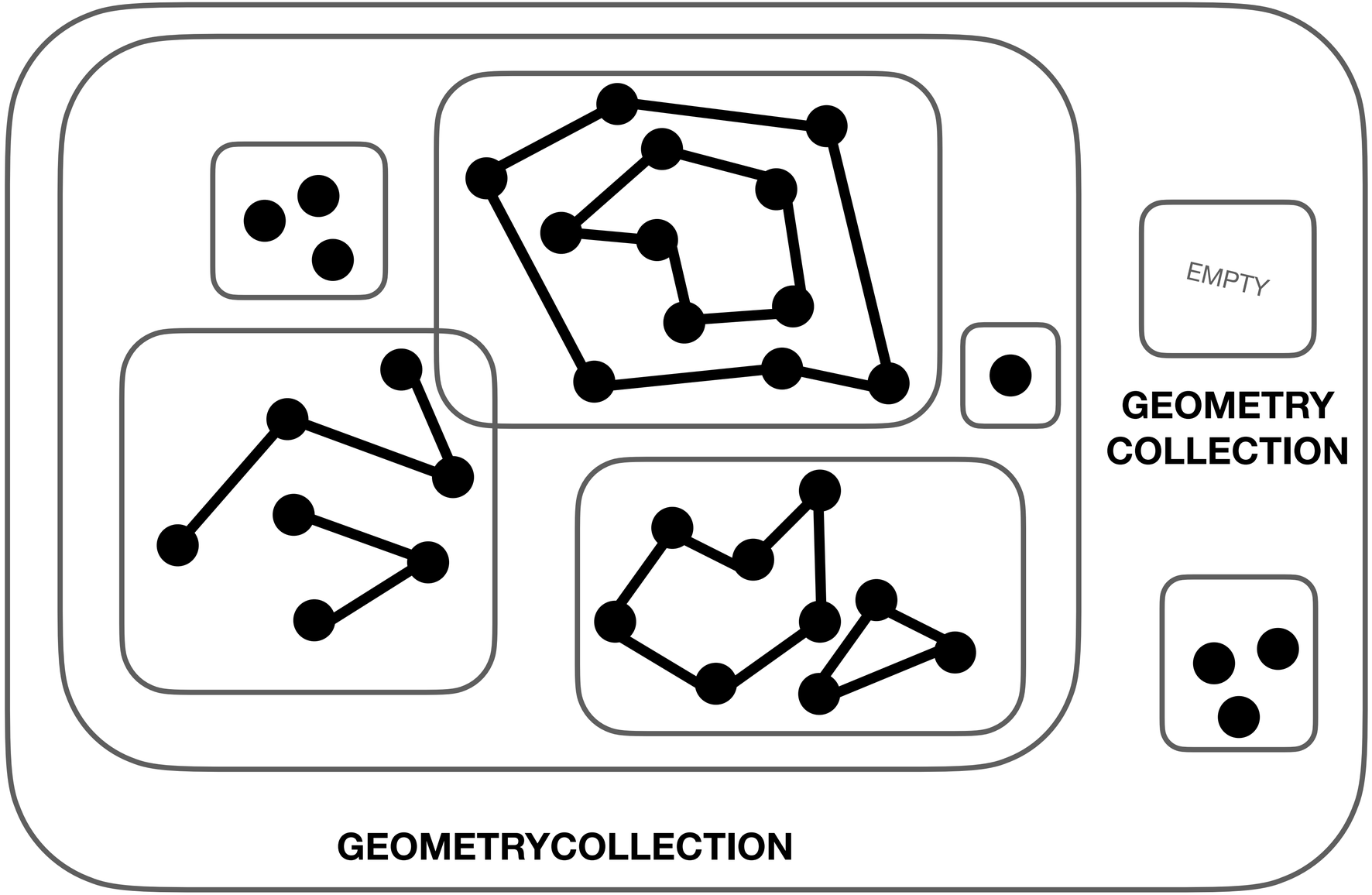
duckdb.org

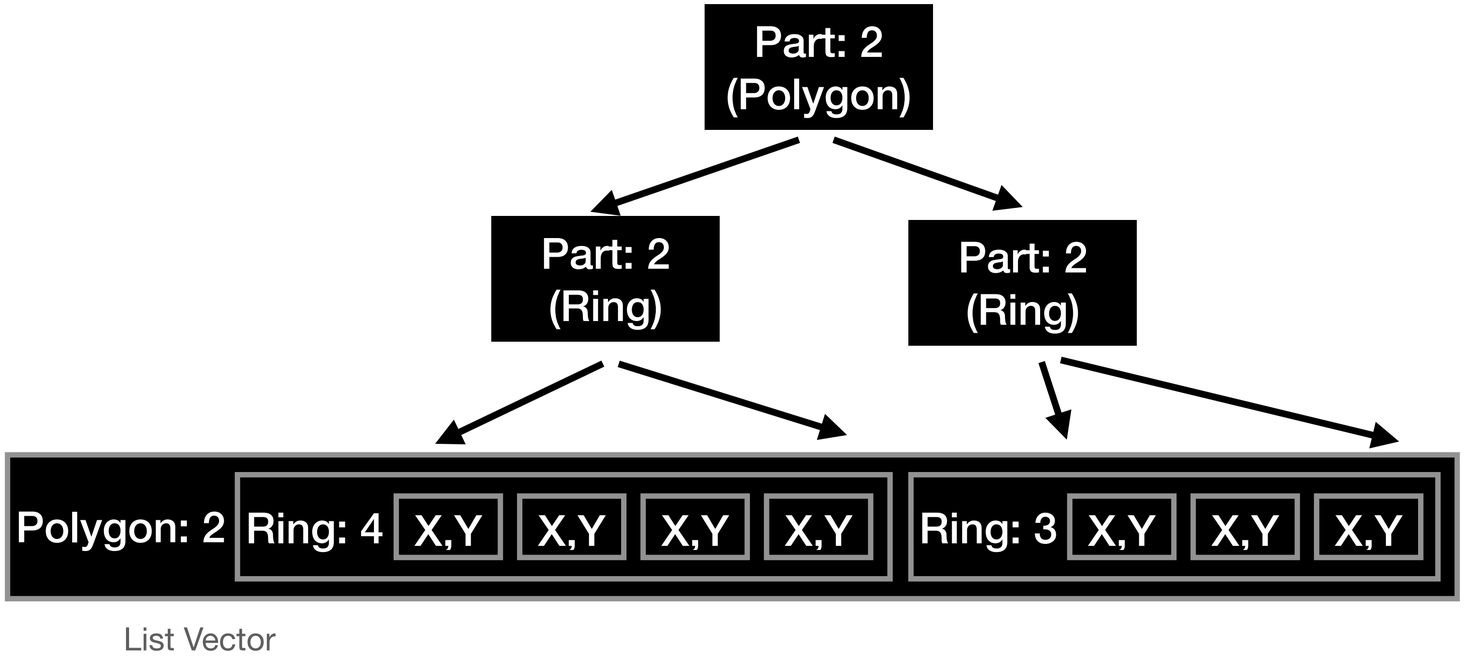
duckdb/duckdb_spatial

x.com/Maxxen_

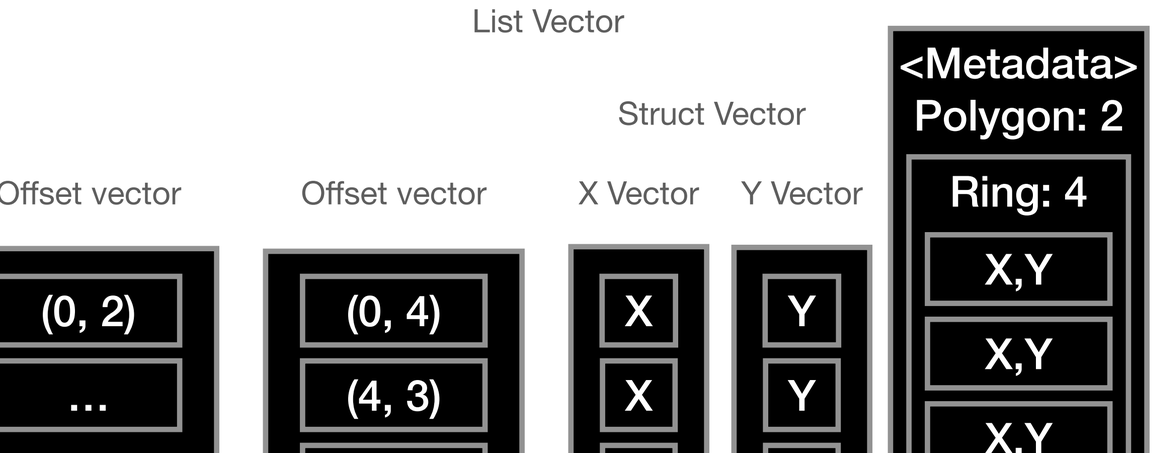


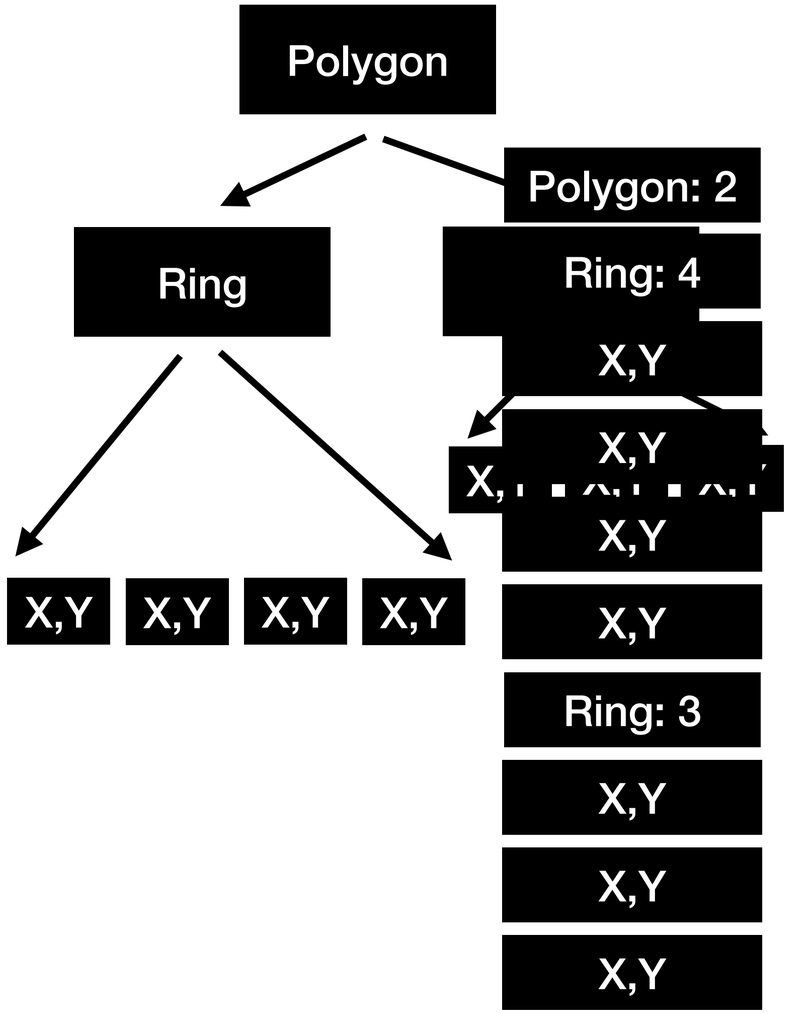
Simple Features




Part: 2
(Polygon)
Part: 2
(Ring)
Part: 2
(Ring)
Polygon: 2
Ring: 4
X,Y
X,Y
X,Y
X,Y
Ring: 3
X,Y
X,Y
X,Y

List Vector


List Vector
Struct Vector
Offset vector
Offset vector
X Vector
Y Vector
(0, 2)
...
(0, 4)
(4, 3)
X
X
Y
Y
<Metadata>
Polygon: 2
Ring: 4
X,Y
X,Y
X,Y


Polygon
Polygon: 2
Ring
Ring: 4
X,Y
X,Y
X,Y
X,Y
X,Y
X,Y
X,Y
X,Y
X,Y
Ring: 3
X,Y
X,Y
X,Y

The three sides of DuckDB

Execution

- Vectorized Engine
- Parallelism
- Query Optimizations

Storage

- Single-File Format
- Compression
- Transactions

UX

- Graceful degradation
- “Friendly SQL”
- In-process

End-To-End Query Optimization

- Access to the full “query plan”
- Expression rewriting
- Filter & Projection pushdown
- Subquery Flattening
- Join Ordering

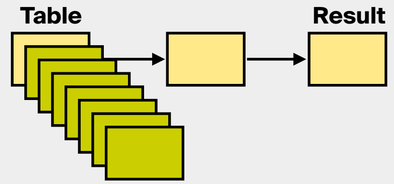
You’d have to do this manually if you write your own transformation pipelines.

But DuckDB will do it for you!

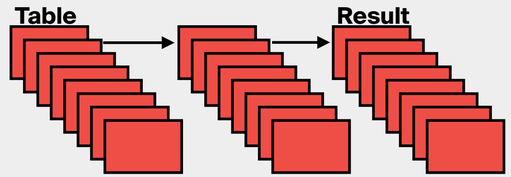
Query Processing

- **Row-at-a-Time**
  - Classic Database approach
  - Low memory footprint
  - High CPU overhead
- **Column-at-a-Time**
  - Common in dataframes
  - Efficient CPU usage, SIMD
  - Large memory footprint
- **Vectorized Processing**
  - Best of both worlds!
  - Optimize for cache locality

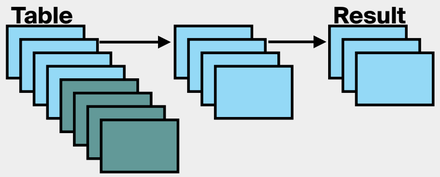
Row-at-a-Time

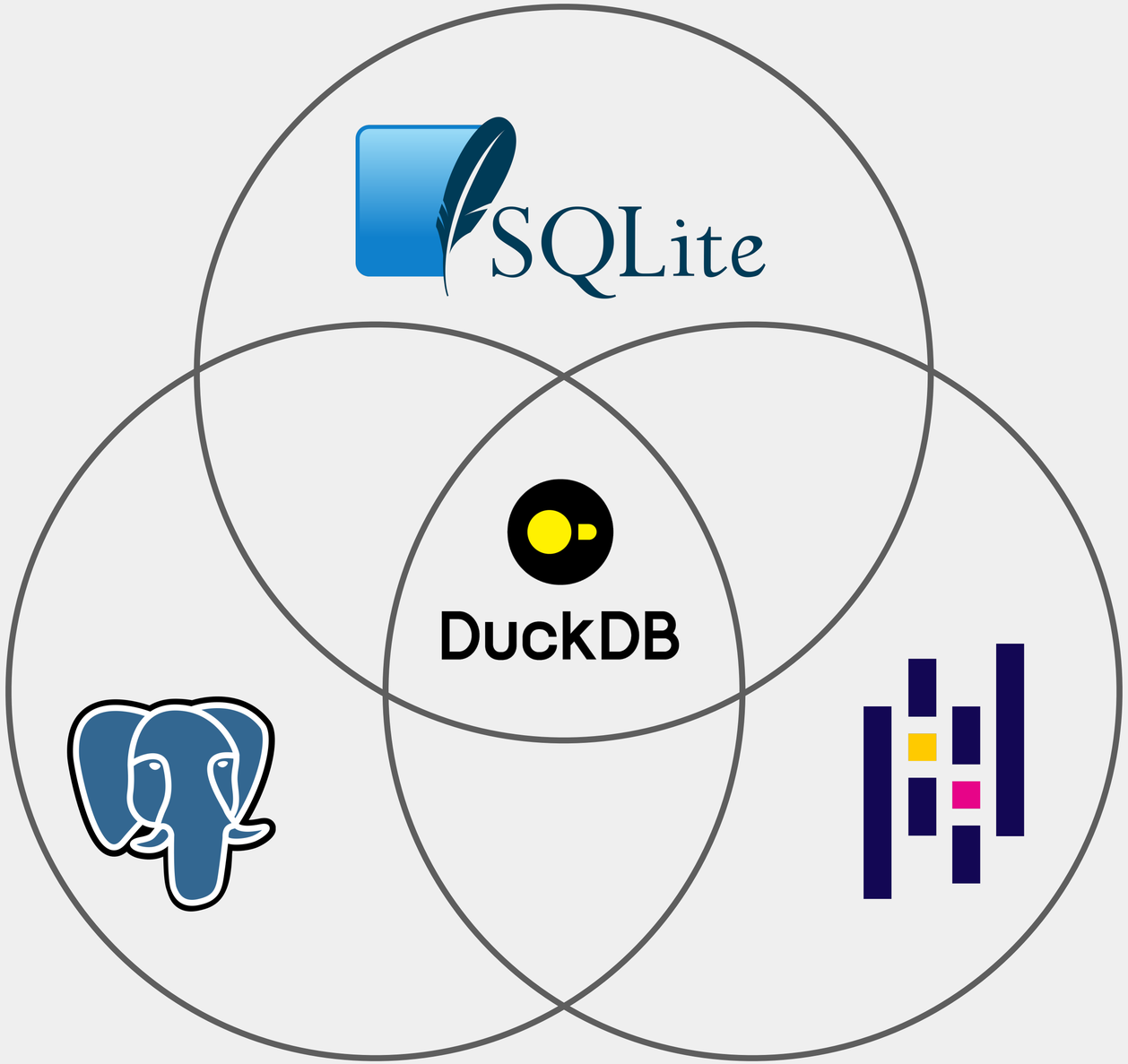


Column-at-a-Time

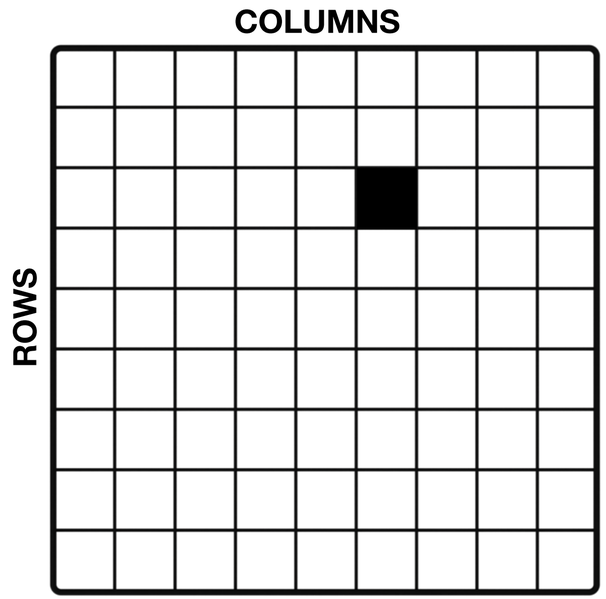


Vectorized Processing

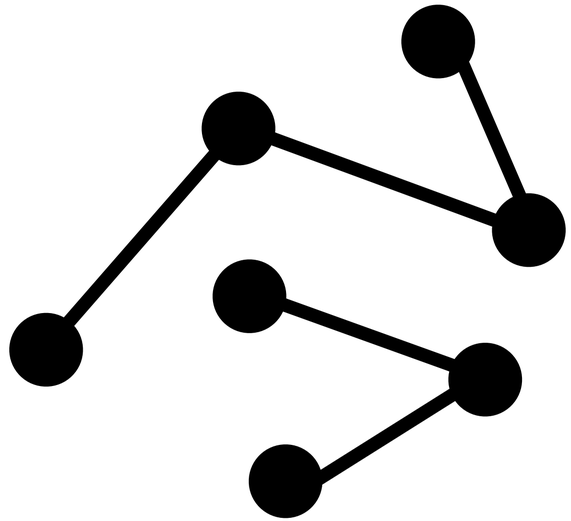



SQLite
DuckDB

Raster vs Vector

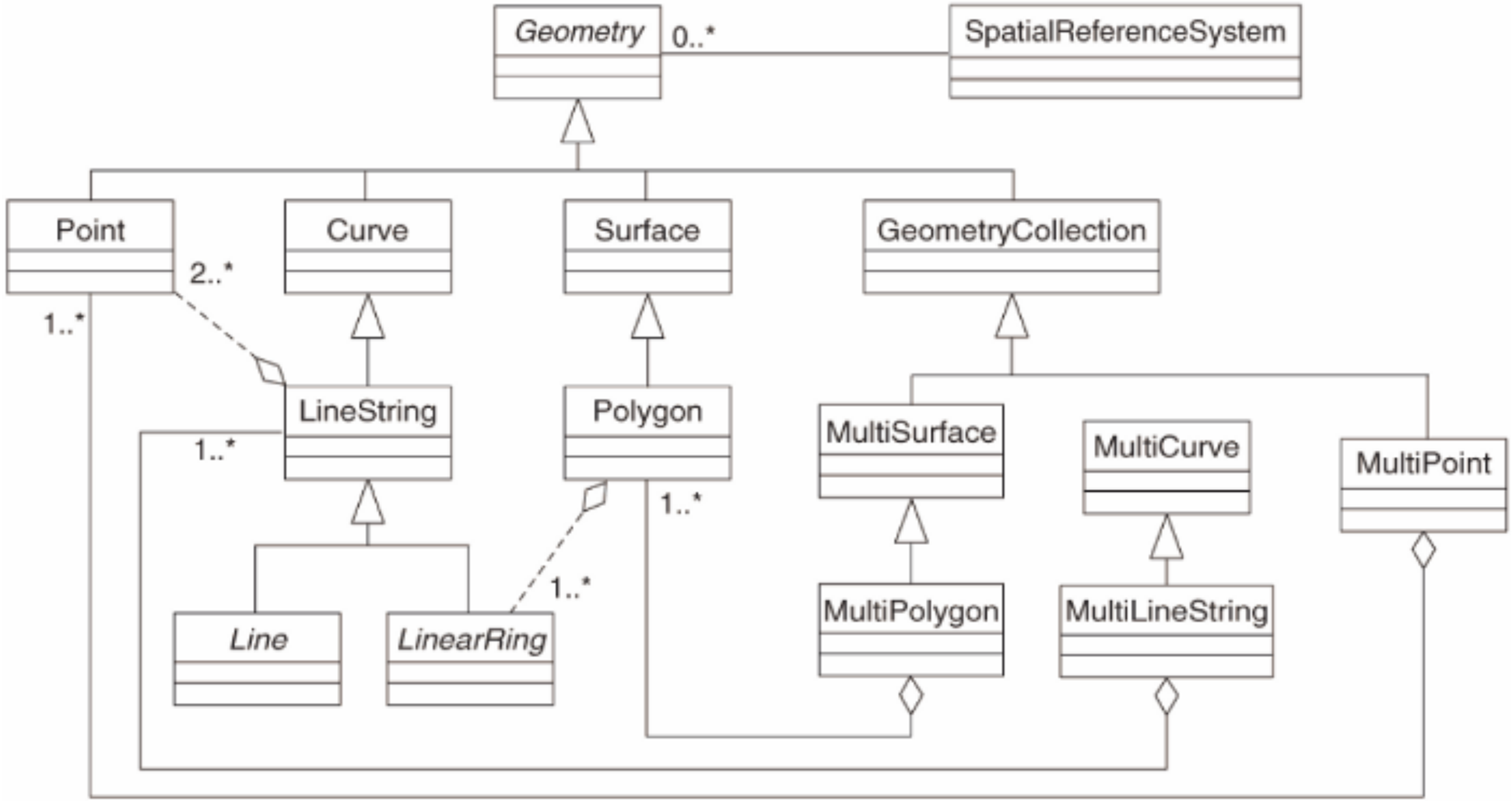


RASTER



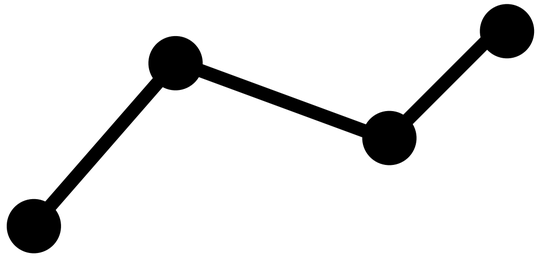
VECTOR

Simple Features

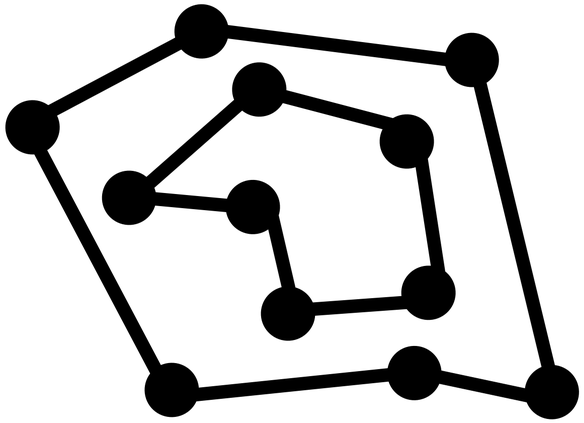


Simple Features

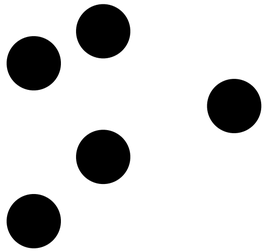
POINT



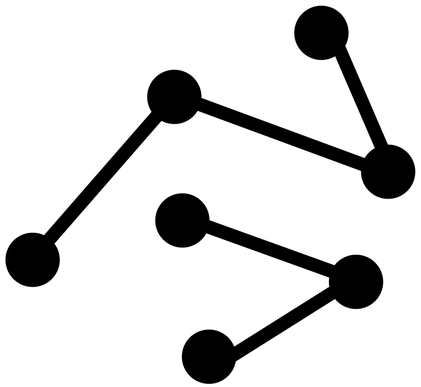
LINESTRING



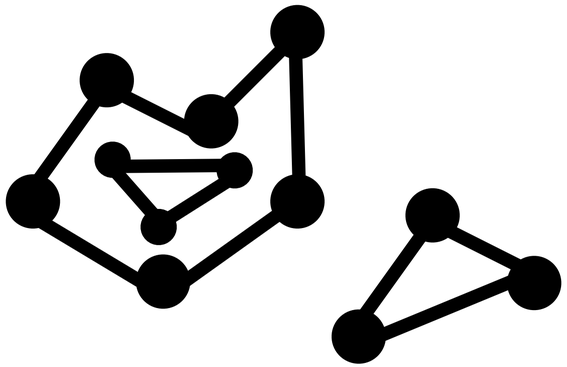
POLYGON



MULTIPOINT

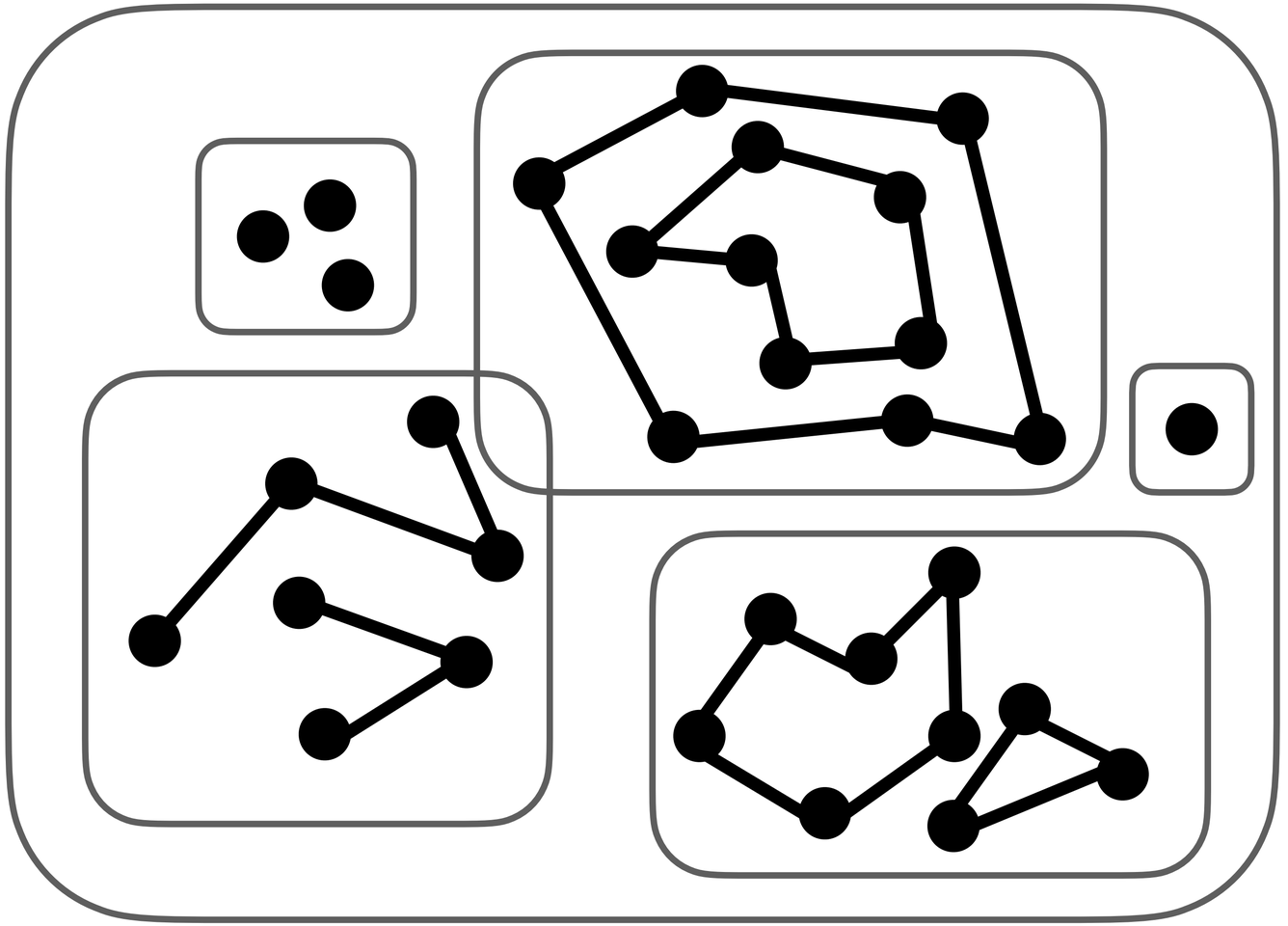


MULTILINESTRING



MULTIPOLYGON

Simple Features



GEOMETRYCOLLECTION

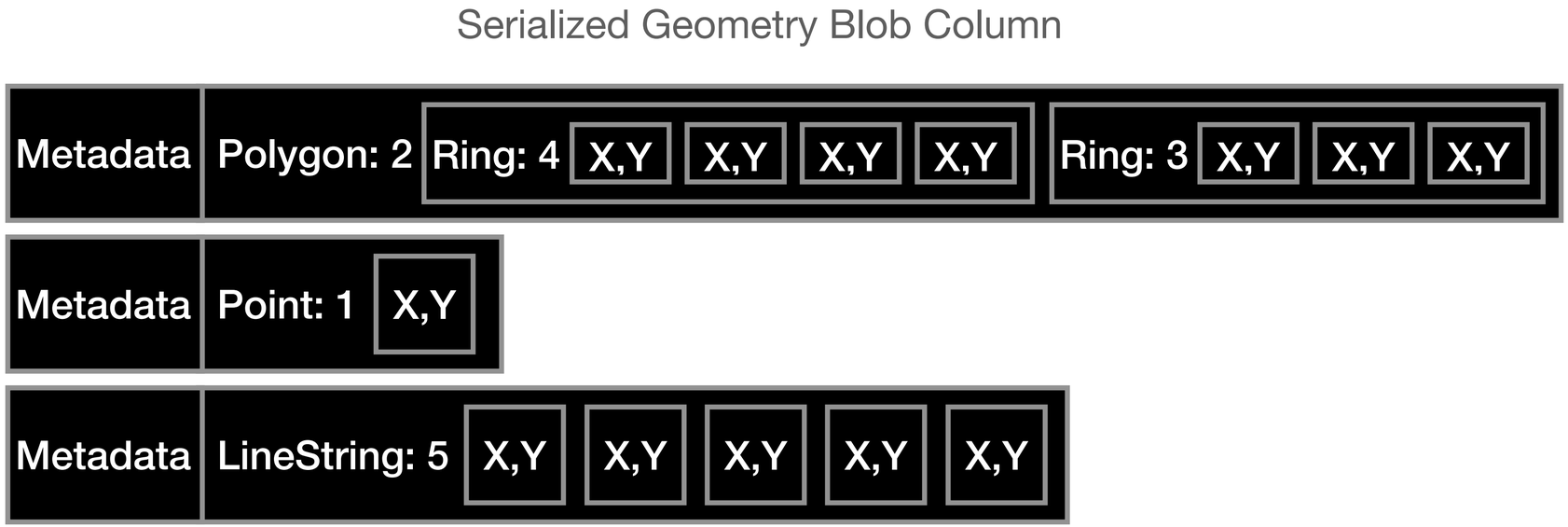
DuckDB Spatial

- Official “Extension” adding geospatial support
- “Simple Features” vector GEOMETRY type
- Modeled after PostGIS, 100+ “ST_” functions
- Statically bundles GDAL/GEOS/PROJ
  - Embedded projection database
  - Export/Import to 40+ vector formats
  - Pre-built binaries for 10 platforms

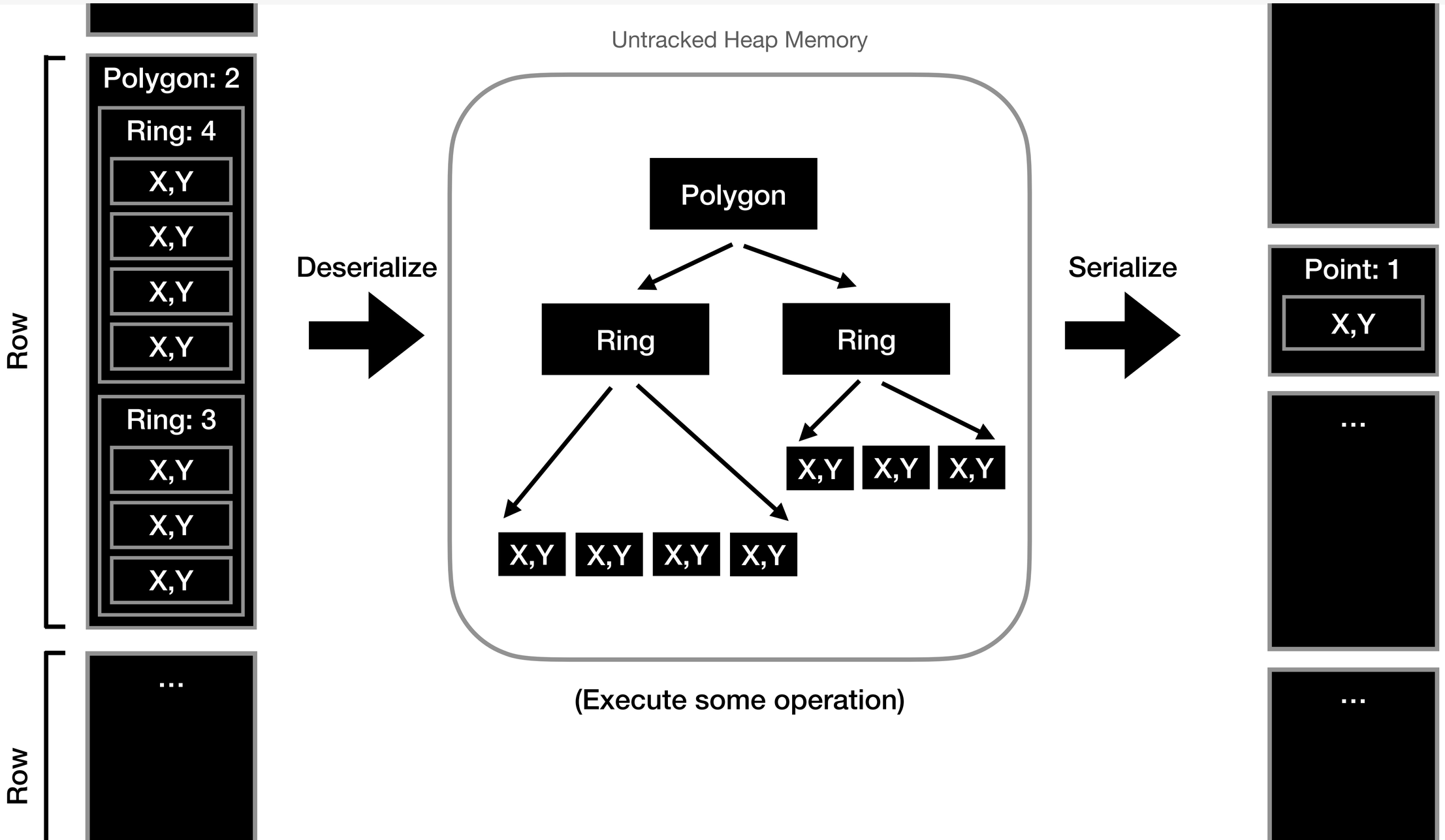
Demo

Challenges

- Geometries require lots of memory
- Compounded by vectorized execution!
- Disk-spilling/storage requires de/serialization
- Store as “blob” due to recursion (GeometryCollection)



Challenges: De/serialization

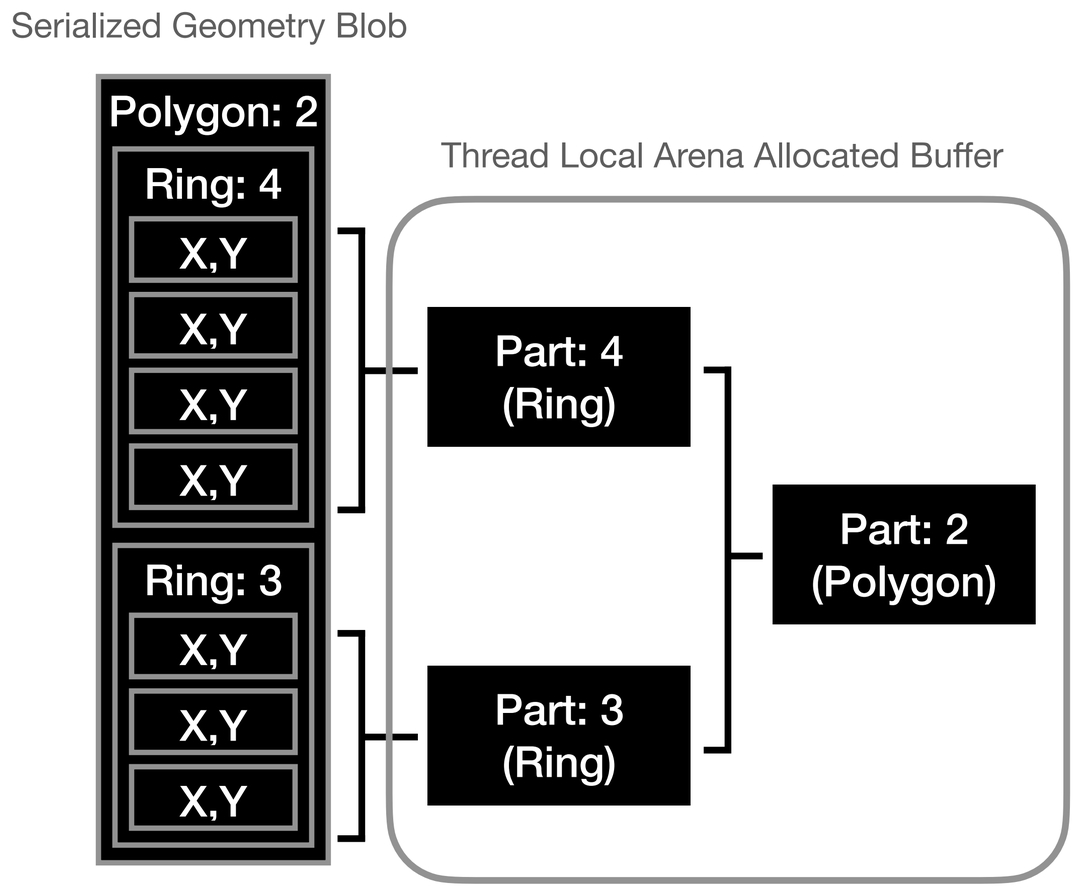


Challenges

- Hard to track memory usage in dependencies
- Revert to row-at-a-time?
  - Lose vectorized efficiency :(
  - Still lots of small de/allocations in the hot path
  - Can’t reuse intermediates
    - Buffers, Edge lists, indexes, etc.
- Piece-wise replace GEOS with “native” representation/algorithms

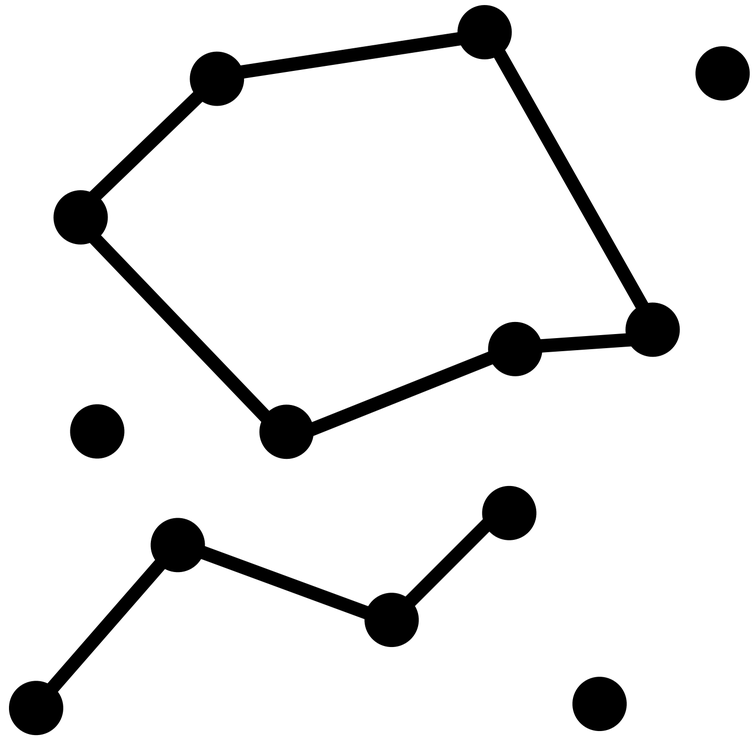
Native Geometry Processing

- **Alt 1:** “Materialized Geometry”
  - { Type, Count, Child Pointer }
  - Arena allocation, Reference vertices
- **Alt 2:** Scan serialized geometries as is!
  - Event based, need to keep state
  - Suitable for simpler properties
- Same problem as XML/JSON parsing
  - DOM vs SAX!



Simple Features from an implementation perspective

- Widely adopted, flexible, but...
  - Untyped, combinatorial explosion
    - Lets not forget: xy, xyz, xym, xyzm
  - Lots of branching and pointer-chasing
  - Unbounded recursion
- Mixing types generally complex




Quiz Time!

```
SELECT ST_Centroid(
  ST_GeomFromText(
    'MULTILINESTRING Z ((2 2 8, 4 4 8))'
    )
)
```

PostGIS: POINT(3 3)
GEOS: POINT(3 3)

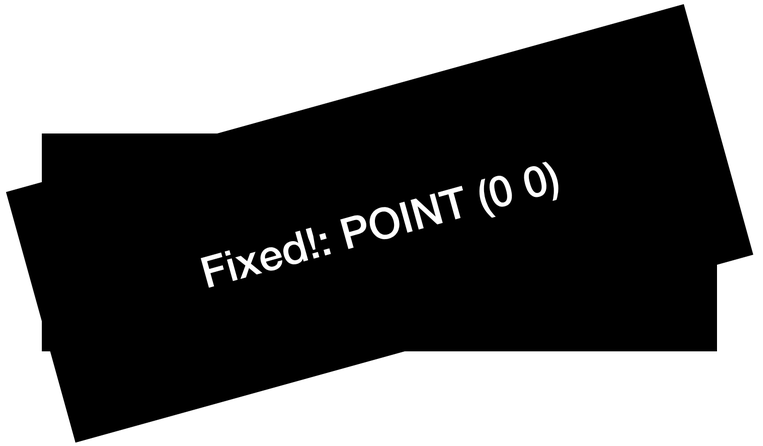
```
SELECT ST_Centroid(
  ST_GeomFromText(
    'MULTILINESTRING ZM EMPTY’
    )
)
```

PostGIS: POINT ZM EMPTY
GEOS: POINT EMPTY

```
SELECT ST_Centroid(
  ST_GeomFromText(
    ‘GEOMETRYCOLLECTION(
      POINT(999 999),
      LINESTRING(0 0, 1 1)
     )’
  )
)
```

PostGIS: POINT (0.5 0.5)
GEOS: POINT (0.5 0.5)

```
SELECT ST_PointOnSurface(
  ST_Collect(
    ST_GeomFromText('LINESTRING(0 0, 1 1)`),
    ST_GeomFromText('LINESTRING EMPTY)`),
  )
)
```

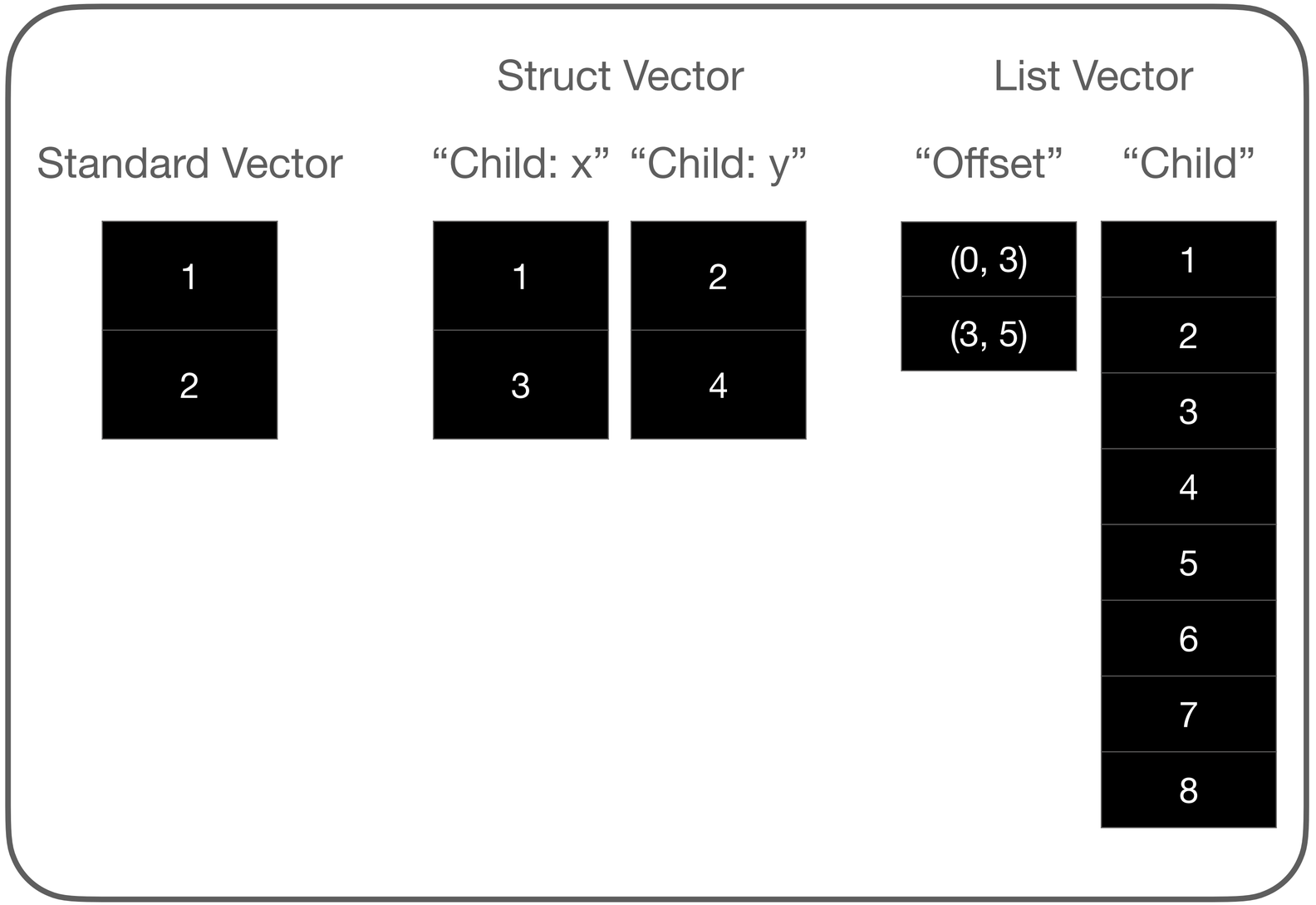


Columnar Geometries

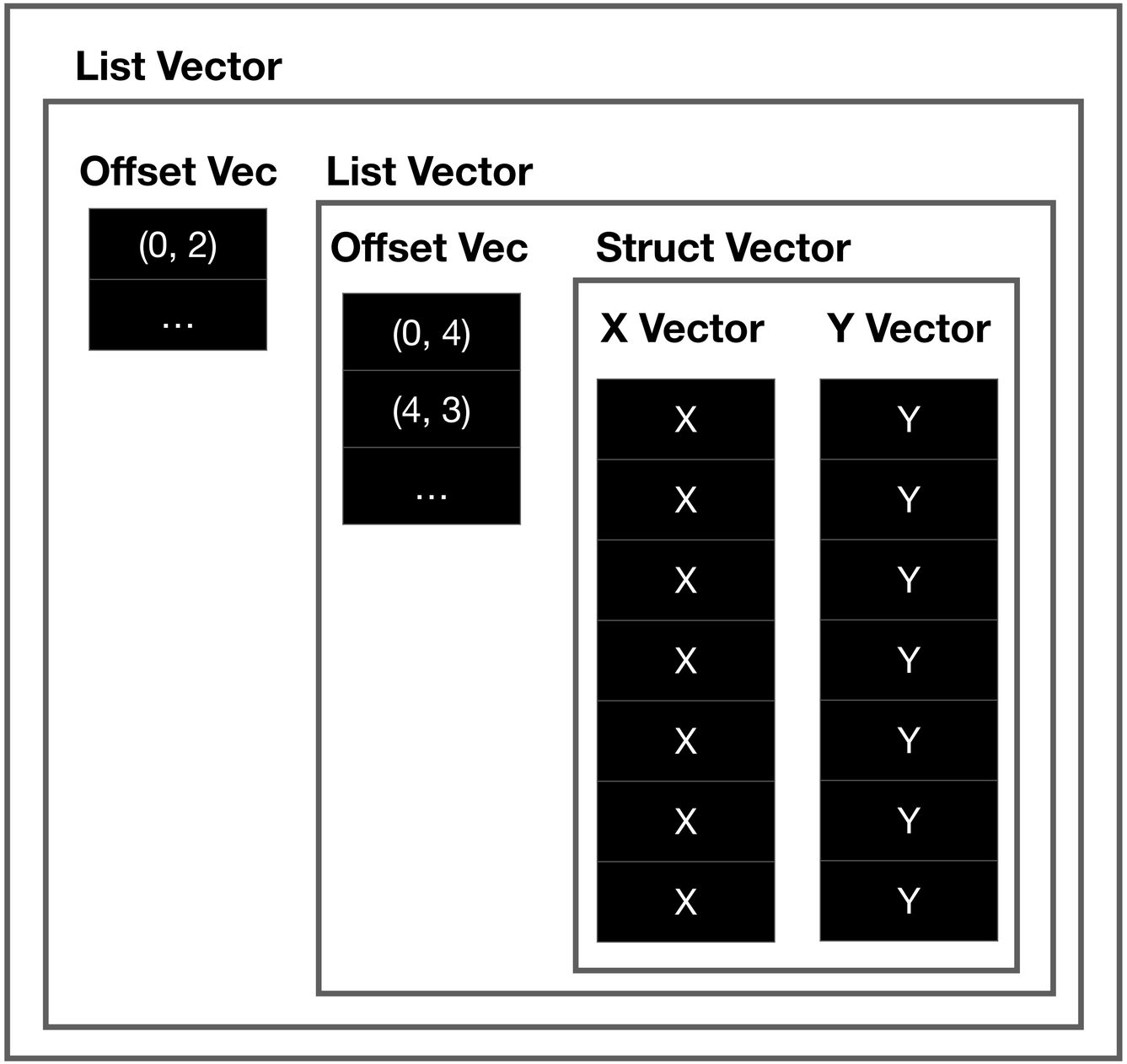
- **Idea:** Store geometry as separate, nested types!

Nested Types in DuckDB

INT	STRUCT(x, y)	LIST(INT)
1	{x: 1, y: 2}	[1, 2, 3]
2	{x: 3, y: 4}	[4, 5, 6, 7, 8]
…	…	…



Nested-Nested Types in DuckDB



Columnar Geometries

- **Idea:** Store geometry as separate, nested types!
- POINT_2D, LINESTRING_2D, POLYGON_2D
  - Not fully parity with GEOMETRY
- No serialization overhead!
- No branching!
- Benefit from built-in statistics/compression!

GeoArrow

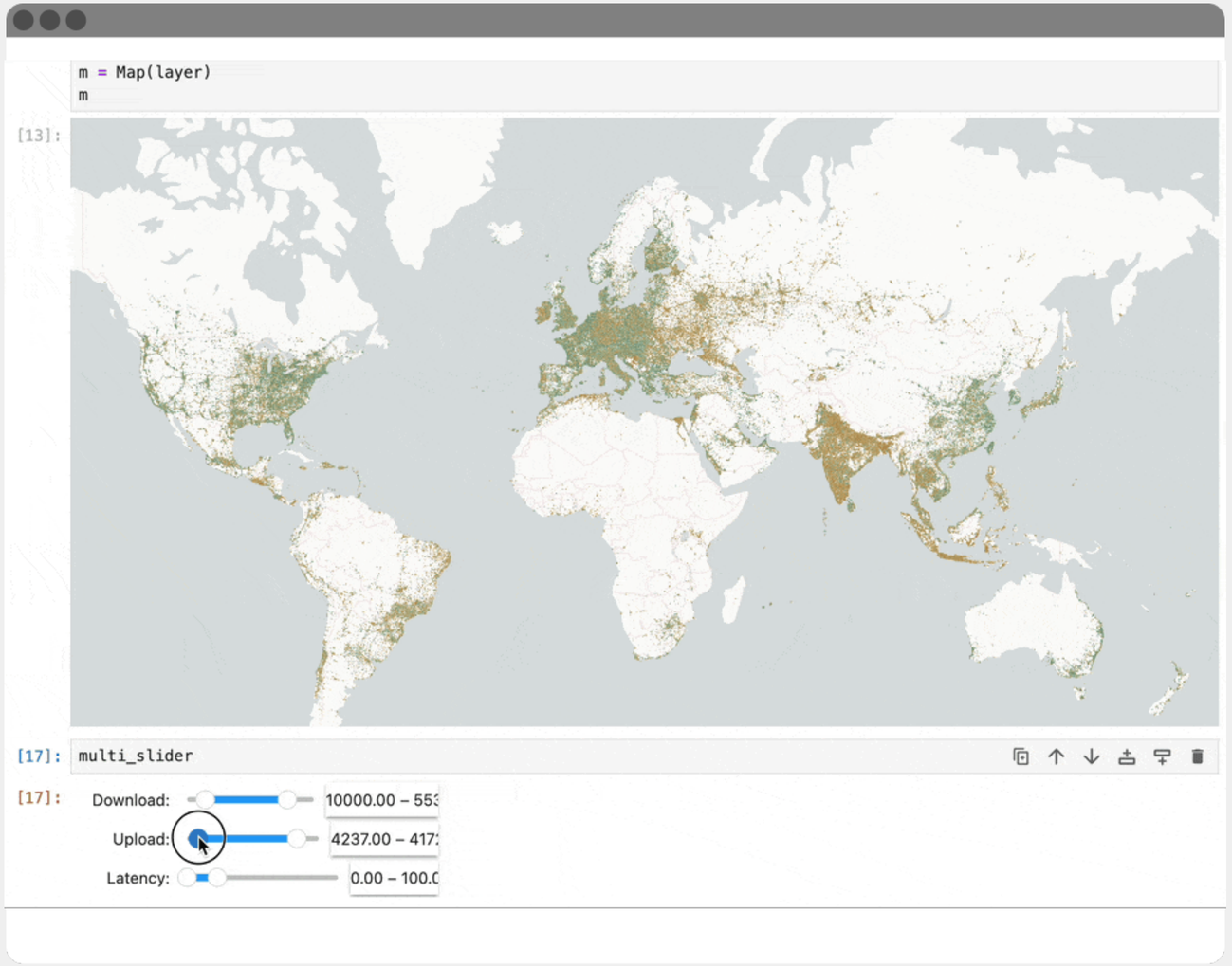
- Spec for arrow-encoded geometry
- Great for visualization, also GPU

- DuckDB doesn’t use Arrow internally
  - But is (roughly) layout compatible!
  - Same goes for spatial
  - Hopefully better interop on the way

GeoArrow

Lonboard

- Plot massive amount of features
- GPU Acceleration
- Browser based w/ deck.gl



https://developmentseed.org/lonboard/latest/ecosystem/duckdb/

And so much more!

- Lots of ecosystem integration
- Support for python UDF’s w/PyArrow
- SQLAlchemy driver
- DBT connector
- IBIS (default backend)
- FSSPEC support