DuckDB: Bringing analytical SQL directly to your Python shell

Pedro Holanda
pedro@duckdblabs.com
pedroholanda.com
Outline

• What is DuckDB?
  • Motivation
  • Main Characteristics

• DuckDB in the Python-Land

• Demo 7~10 minutes.
  • Estimating NYC taxi fare costs with DuckDB, Pandas and PySpark.

• Summary
What is DuckDB
Motivation

- Combining Database Management Systems with Data Science
- DB Connection
- Embedded

[Diagram showing integration of database and analytics systems]
Main Characteristics

▸ DuckDB: The SQLite for Analytics

▸ Simple installation
  $ pip install duckdb

▸ Embedded: no server management

▸ Fast analytical processing

▸ Fast transfer between R/Python and RDBMS

▸ DuckDB is currently in pre-release (V0.6)
  Check duckdb.org for more details.
Main Characteristics

- Columnar Data Storage
- Vectorized Execution Engine
- End-to-end Query Optimization
- Automatic Parallelism
- Data Compression
- Beyond Memory Execution
Columnar Data Storage

- **Row-Storage:**
  - Individual rows can be fetched cheaply.
  - However, all columns must always be fetched!
- What if we only use a few columns?
- e.g.: What if we are only interested in the price of a product, not the stores in which it is sold?
Columnar Data Storage

- **Column-Storage:**
  - We can fetch individual columns
  - Immense savings on disk IO/memory bandwidth when only using few columns
Compression

- Individual columns often have similar values, e.g. dates are usually increasing
- Save ~3-5X on storage (depending on compression algorithms used and data)

<table>
<thead>
<tr>
<th>DuckDB Version</th>
<th>Taxi</th>
<th>Ratio</th>
<th>Lineitem</th>
<th>Ratio</th>
<th>Compression</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2.8</td>
<td>15.3 GB</td>
<td>1</td>
<td>0.85 GB</td>
<td>1</td>
<td>None</td>
<td>07/21</td>
</tr>
<tr>
<td>0.2.9</td>
<td>11.2 GB</td>
<td>1.36x</td>
<td>0.79 GB</td>
<td>1.07x</td>
<td>RLE + Constant</td>
<td>09/21</td>
</tr>
<tr>
<td>0.3.2</td>
<td>10.8 GB</td>
<td>1.41x</td>
<td>0.56 GB</td>
<td>1.51x</td>
<td>Bitpacking</td>
<td>02/22</td>
</tr>
<tr>
<td>0.3.3</td>
<td>6.9 GB</td>
<td>2.21x</td>
<td>0.32 GB</td>
<td>2.64x</td>
<td>Dictionary</td>
<td>24/22</td>
</tr>
<tr>
<td>0.5.0</td>
<td>6.6 GB</td>
<td>2.31x</td>
<td>0.29 GB</td>
<td>2.93x</td>
<td>For</td>
<td>09/22</td>
</tr>
<tr>
<td>0.6</td>
<td>4.8 GB</td>
<td>3.18x</td>
<td>0.17 GB</td>
<td>5x</td>
<td>FSST + CHIMP</td>
<td>11/22</td>
</tr>
</tbody>
</table>
Compression

- **Example:**
  We have a query that requires 5 columns of the table.

- **No compression:**
  Read 5 columns (50GB) from disk ≈ 8 minutes

- **Compression:**
  Read 5 compressed columns (5x = 10GB) from disk ≈ 1:40 minutes
SQLite use tuple-at-a-time processing
- Process one row at a time

Pandas use column-at-a-time processing
- Process entire columns at once

DuckDB uses vectorized processing
- Process batches of columns at a time
Execution

- **Tuple-at-a-Time (SQLite)**
  - Optimize for low memory footprint
  - Only need to keep **single row** in memory
  - Comes from a time when **memory was expensive**
  - **High CPU overhead per tuple!**
Execution

- **Column-at-a-Time (Pandas)**
  - Better CPU utilization, allows for SIMD
  - Materialize **large intermediates** in memory!
  - Intermediates can be gigabytes each...
  - **Problematic** when data sizes are large
Vectorized Processing (DuckDB)

- Optimized for CPU Cache locality
- SIMD instructions, Pipelining
- Small intermediates *(ideally fit in L1 cache)*
End-To-End Query Optimization

- Expression rewriting
- Join Ordering
- Subquery Flattening
- Filter/Projection Pushdown
Automatic Parallelism & Beyond Memory Execution

- **DuckDB** has parallel versions of most operators
  - Scanners (Insertion Order Preservation)
  - Aggregations
  - Joins
- **Pandas** only support single-threaded execution.

- DuckDB supports execution of data that does not fit in memory
  - Graceful Degradation
  - Never Crash always executes query
DuckDB  In the Python Land
APIs

- **Python DB API 2.0 Compliant**

```python
import duckdb
con = duckdb.connect("duck.db")
con.execute("SELECT j+1 FROM integers WHERE i=2")
```

- **Relational API**

```python
import duckdb
con = duckdb.connect("duck.db")
# Table operator returns a table scan
rel = con.table("integers")
# We can inspect intermediates
rel.show()
# We can chain multiple operators
rel.filter("i=2").project("j+1").show()
```
Integrations

- Tight Integration - Zero Copy (Input + Output)

  **Pandas**

  ```python
  import pandas as pd
  import duckdb
  d = {"col1": [1, 2], "col2": [3, 4]}
  df = pd.DataFrame(data=d)
  con = duckdb.connect()
  # Consumes Pandas Dataframe
  res = con.execute("select * from df")
  # Produces Pandas Dataframe
  result_dataframe = res.df()
  ```

  **PyArrow**

  ```python
  import pyarrow as pa
  import duckdb
  d = {"col1": [1, 2], "col2": [3, 4]}
  arrow = pa.Table.from_pydict(d)
  con = duckdb.connect()
  # Consumes Arrow Object
  res = con.execute("select * from arrow")
  # Produces Arrow Table
  result_array = res.array()
  ```

- NumPy

- SQL Alchemy

- IBIS (Default Backend)
Usage

Downloads last day: 33,594
Downloads last week: 251,770
Downloads last month: 898,816

Daily Download Quantity of duckdb package - Overall

Downloads

Date


With_Mirrors
Without_Mirrors
Summary
DuckDB is an embedded database system.

Designed for **Analytical Queries** (i.e., Data Analysis/Science).

**Open-Source** (Under MIT license) and free to use!

Has **binding for many languages** (e.g., Python, R, Java...)

**Tightly integrated** with the Python Ecosystem.

**Zero-Copy** access to Python/NumPy and PyArrow datasets.

Implements the **DB and Relational APIs**.

**Full SQL** Support!

Pedro Holanda
pedro@duckdblabs.com
pedroholanda.com