ChatGPT

DuckDB Python Client Advanced Reference (v1.4+)

This reference guide covers the advanced usage of DuckDB's Python client (DuckDB ≥ **1.4**), focusing on modern APIs and features. It is organized for expert Python developers and AI agents, emphasizing clarity, completeness, and ready-to-run examples. All examples assume `import duckdb` (and other relevant libraries) has been done.

Connection and Basic Usage

DuckDB's Python API primarily uses a **connection object** (class `DuckDBPyConnection`). You create a connection with `duckdb.connect(...)` [1], which opens or creates a DuckDB database (in-memory or file):

```
import duckdb
# In-memory DB (new for each unnamed :memory:)
conn = duckdb.connect(database=':memory:')
# Named in-memory DB (shared if name reused)
conn_shared = duckdb.connect(database=':memory:my_db')
# Persistent database file (creates file if not exists)
conn_file = duckdb.connect('analytics.duckdb')
# Read-only mode (e.g., for a file)
conn_ro = duckdb.connect('analytics.duckdb', read_only=True)
```

- Using the special string `':memory:'` opens an ephemeral in-memory database [2]. A **named** in-memory DB (e.g. `:memory:my_db`) can be re-used by future connections with the same name [3]. The special name `':default:'` refers to the module's **default connection** (opened implicitly on first use) [4].
- If a filesystem path is given (e.g. `"analytics.duckdb"`), DuckDB opens a persistent database at that location [5] (creating it if needed). Use `read_only=True` to prevent modifications [6].
- A connection can be closed with `conn.close()`. When the last connection to a DB is closed, the database is shut down [7].

Each `DuckDBPyConnection` supports the standard **DB-API** cursor methods directly (the connection object acts as its own cursor). You can execute SQL queries using `execute()` or the shorthand `sql()` method, and retrieve results with fetch methods [8]:

```
result = conn.execute("SELECT 'Hello, DuckDB' AS msg;")  # Execute SQL
print(result.fetchone())   # Fetch one result row
```

```
# Alternatively, chain calls:
print(conn.execute("SELECT 42").fetchall())  # [(42,)]
```

- **Auto-commit**: By default, each SQL statement commits immediately (DuckDB uses single-statement transactions). You can group statements in a transaction using BEGIN/COMMIT SQL or the `conn.begin()` and `conn.commit()` methods if needed.
- **Multiple cursors**: DuckDB allows **multiple connections** to the same database file (one writer & multiple readers). Within one process, you can call `conn.cursor()` to get an additional handle on the same connection (useful for parallel threads) [9]. However, a single connection is thread-safe and will serialize queries if used from multiple threads.
- **Default connection shortcut**: The top-level `duckdb` module functions (like `duckdb.sql()` or `duckdb.query()`) operate on a hidden default connection [10]. This means you can use DuckDB without explicitly managing a connection for simple use cases, but for clarity and resource management, using an explicit connection is recommended in long-lived programs.

Lazy Relational API and Query Execution

DuckDB's Python client offers a **Relational API** that allows building queries step by step using `Relation` objects. This API enables **lazy execution**: no SQL is actually run until you request results [11] [12].

For example, using `conn.sql()` (or `conn.from_*` methods) returns a `DuckDBPyRelation`:

```
rel = conn.sql("SELECT * FROM range(1000000000)")  # 1 billion rows, not yet
executed
print(type(rel))            # DuckDBPyRelation
# No data has been materialized yet (lazy).
```

At this point, `rel` is a symbolic representation of the query and holds **no data in Python** [11]. Only when an **execution trigger** is called will DuckDB execute the query. Triggers include:

- Calling `rel.fetch…()` or conversion methods like `rel.df()` (to pandas) or `rel.fetchall()` – these execute the query and bring results into Python.
- Printing or showing the relation (e.g. `rel.show()`) – this will execute and display up to a limit (10k rows by default) [12].
- Saving or creating a table/view from the relation (e.g. `rel.create('table_name')` or `rel.to_parquet('file.parquet')`).

For example, `rel.show()` would execute the query and print the first few rows:

```
rel = conn.sql("SELECT * FROM range(1000000)")
rel.show()  # triggers execution, prints up to 10000 rows [12]
```

DuckDB prints a truncated result if there are more than 10k rows, indicating how many rows were not shown [12]. You can also explicitly convert or fetch all data (see next section).

The Relational API allows **method chaining** to build complex queries without writing SQL. Common methods include:

- `relation.filter("condition")` – apply a `WHERE` filter [13].
- `relation.project("expressions")` – select or compute specific columns (similar to `SELECT` list).
- `relation.join(other_relation, "condition")` – join with another relation (uses SQL join semantics).
- `relation.aggregate("agg_expr", "group_expr")` – aggregation with GROUP BY.
- `relation.order("order_expr")` – sort the result.
- ...and more (these correspond to SQL operations). For example:

```
rel = conn.from_df(df) \
        .filter("score > 50") \
        .project("user, score*1.1 as adjusted_score") \
        .order("adjusted_score DESC")
```

None of these operations hit the database until you request output. This lazy approach enables DuckDB to optimize the query as a whole, potentially improving performance.

If you prefer, you can always execute raw SQL directly using `conn.execute()` / `conn.sql()` and skip the relation API. The relation API is especially handy for incremental query construction or when integrating with pandas/Polars.

Tip: You can create relations from various sources: - `conn.table("existing_table")` – reference a table in the DB [14]. - `conn.view("my_view")` – reference a SQL view [15]. - `conn.from_df(pandas_df)` – create relation from a Pandas DataFrame in memory [16]. - `conn.from_arrow(pyarrow_table)` – from a PyArrow Table/RecordBatch [17]. - `conn.from_parquet('file_or_glob.parquet')` – from Parquet file(s) [18]. - `conn.from_csv_auto('file.csv')` (alias `read_csv`) – from CSV with auto-detection [16]. - ... etc. All these create a lazy `Relation` that you can further refine.

Converting Query Results to DataFrames, Arrow, and NumPy

When you need to retrieve results from DuckDB into Python objects, the Python client provides a rich set of **result conversion functions**. These functions **execute the pending query** (if not already executed) and pull the data into the requested format:

- **Pandas DataFrame**: Use `relation.df()` or `relation.to_df()` to get a `pandas.DataFrame` [19] [20]. For example:

```
df = conn.sql("SELECT * FROM my_table").to_df()  # Execute and fetch all
into pandas
```

Aliases: `fetchdf()` and `fetchall()` can also return a DataFrame by default [19].

- **Polars DataFrame**: Use `relation.pl()`. Example:

```
import polars as pl
pl_df = conn.sql("SELECT 1 AS id, 'apple' AS fruit").pl()
# pl_df is a polars.DataFrame
```

By default this returns a Polars `DataFrame` [21] [22]. You can also get a Polars **LazyFrame** with `relation.pl(lazy=True)` [23], which gives a deferred Polars query (useful for further Polars operations without materializing data).

- **Arrow Table**: Use `relation.arrow()` or `relation.fetch_arrow_table()`. This returns a PyArrow Table of all results [24]. You can also get a streaming Arrow **RecordBatchReader** with `relation.fetch_arrow_reader()` [24]. For example:

```
import pyarrow as pa
arrow_table = conn.sql("SELECT * FROM large_table").arrow()  #
pyarrow.Table
```

DuckDB's Arrow integration is zero-copy; it efficiently converts data to Apache Arrow format [25] [26].

- **NumPy**: Use `relation.fetchnumpy()` to get a `dict` of column name -> numpy.ndarray [27]. For example:

```
data = conn.sql("SELECT x, y FROM values (1,10), (2,20)").fetchnumpy()
# data is {'x': array([1, 2]), 'y': array([10, 20])}
```

This is convenient for interoperability with machine learning libraries expecting NumPy arrays.

- **PyTorch / TensorFlow**: For machine learning workflows, `relation.torch()` returns a dict of PyTorch tensors, and `relation.tf()` returns a dict of TensorFlow tensors [28] [29]. (These require the respective libraries to be installed.)

- **CSV or Parquet file**: You can directly write out results without returning them to Python memory:

- `relation.to_parquet("file.parquet")` writes the query result to a Parquet file (or multiple files if the relation is partitioned) [30]. For example:

```
conn.sql("SELECT * FROM big_table").to_parquet("big_table.parquet")
```

Additional options (compression, partitioning, etc.) can be provided as arguments [31].

- `relation.to_csv("file.csv")` similarly writes CSV. Aliases: `write_parquet()` and `write_csv()` can be used interchangeably.

- **Materialize as Table/View**: Use `relation.create("table_name")` to create a persistent table in the database from the relation's result [32]. Use `relation.create_view("view_name")` or `to_view()` to register a logical view [33]. For example:

```
conn.sql("SELECT * FROM myview").create("new_table")  # new_table now
exists in DuckDB
```

All the above methods trigger the query execution if it wasn't already run, due to DuckDB's lazy evaluation model [11]. Be mindful of memory: fetching a huge result into pandas/Polars might be slow or use a lot of RAM. For large datasets, it's often better to keep the data in DuckDB or use Arrow/PyTorch readers to stream data in manageable batches.

Reading and Writing Data Files (CSV/Parquet/JSON and More)

DuckDB can directly read and write a variety of data formats, both local and remote, with minimal setup:

- **CSV Files**: Use `duckdb.read_csv("file.csv")` to lazily read a CSV into a relation [34]. You can pass wildcards to read multiple files (e.g. `"data/2023/*.csv"` to read a folder) [35]. The CSV reader auto-detects delimiters, header, etc., but you can specify options (e.g. `header=False, sep="|"`) [36] or column types (`dtype=["int","varchar"]`, etc.) [37]. In SQL, `SELECT * FROM 'file.csv'` also works (DuckDB will invoke the CSV reader) [38]. For example:

```
# From Python API:
rel = duckdb.read_csv('records.csv', header=True, sep=',')
# From SQL via the connection:
conn.sql("SELECT count(*) FROM 'records.csv'").fetchall()
```

  Writing CSV: Use `COPY` SQL or `relation.to_csv("out.csv")` to export results to CSV.

- **Parquet Files**: Use `duckdb.read_parquet("file.parquet")` to get a relation over a Parquet file [39]. This also supports wildcards or lists of files [40]. For example, `duckdb.read_parquet("s3://my-bucket/data/part_*.parquet")` (with appropriate S3 configuration) will query multiple Parquet files as one table. You can also query Parquet directly in SQL (`SELECT * FROM 'file.parquet'`) [41]. DuckDB can **write Parquet** using the `COPY` command or `relation.to_parquet()` as noted. For instance:

```
conn.execute("COPY my_table TO 'backup.parquet' (FORMAT PARQUET)")
```

to export a table to Parquet. DuckDB's Parquet reader is highly optimized, and you can even query Parquet files **in-place** without importing them (DuckDB will only read relevant columns and row groups).

- **JSON Files**: DuckDB (with the JSON extension, which is built-in by default) can read JSON. Use `duckdb.read_json("file.json")` for auto-detecting JSON format (normal or newline-delimited) [42]. Wildcards are supported for batch loading [43]. In SQL, use `read_json('file.json')` table function or simply query `'file.json'` (DuckDB will auto-detect via extension) [44]. For large JSON, DuckDB infers a schema and can query JSON data without fully loading it into memory. Writing JSON is possible via `COPY TO ... (FORMAT JSON)`.

- **Excel Files**: With the `excel` extension, DuckDB can read `.xlsx` files. After installing & loading the extension (see **Extensions** section), you can do:

```
SELECT *
FROM read_xlsx('report.xlsx', sheet='Sheet1');
```

  or simply `SELECT * FROM 'report.xlsx'` [45]. The `read_xlsx` table function allows specifying `sheet=` name, or a cell `range='A1:D100'` to limit the import [46] [47]. Only `.xlsx` (Excel OpenXML) is supported, not old `.xls` [48]. You can also export tables to Excel using `COPY table TO 'out.xlsx' (FORMAT XLSX)` [49].

- **Other formats**: DuckDB supports reading **Apache Iceberg** tables, **Delta Lake** tables, and more via extensions (see **Extensions**). For example, with the `delta` extension loaded, you can query a Delta Lake folder with:

```
SELECT * FROM delta.read_delta('path/to/delta_table');
```

  (DuckDB will handle the Delta transaction log and present the latest table state) [50]. Similarly, the `iceberg` extension allows connecting to Iceberg tables (e.g., after configuring an Iceberg catalog, using `SELECT * FROM my_iceberg_table` in SQL). These advanced formats require proper extension setup.

Local vs Remote Files: DuckDB can read files from local disk, and with the `httpfs` extension (usually included in DuckDB Python), it can access remote URLs (S3, HTTP/HTTPS, Google Cloud Storage, etc.). For example, you can directly query a Parquet over HTTPS:

```
rel = duckdb.read_parquet("https://domain.com/path/data.parquet")  # remote file
```
[51]

If credentials are required (e.g., S3 private bucket), see **Secrets Management** below. The `httpfs` extension supports `s3://`, `gs://`, and `http://` / `https://` URLs natively. By default, DuckDB will download the remote file in chunks on the fly.

- **Caching**: DuckDB can cache remote file data in-memory to avoid repeated downloads. The configuration `enable_external_file_cache` controls this (default `true`) [52]. Additionally, `parquet_metadata_cache` can cache Parquet file metadata (default off) [53]. These improve performance when you query the same remote files multiple times. (Note: A former setting `enable_object_cache` is legacy and inactive [54], so use the new cache settings.)

- **fsspec integration**: In Python, DuckDB also integrates with **fsspec** for filesystems not natively supported by `httpfs`. For example, to read from Google Cloud Storage via `gcsfs` (which is not supported by default `httpfs`), you can register an fsspec filesystem:

```
import fsspec, duckdb
import gcsfs  # ensure GCS filesystem is installed
duckdb.register_filesystem(fsspec.filesystem('gcs'))  # register GCS fs [55]
# Now use the protocol in a read function:
conn.sql("SELECT * FROM read_csv('gcs://my-bucket/path/data.csv')").show()
```

  DuckDB will route the `gcs://` access through fsspec [56] [55]. Keep in mind that fsspec-based access goes through Python and may be slower than DuckDB's native connectors. But it enables a wide range of storage backends.

Querying In-Memory Data (Pandas, Polars, NumPy, Arrow)

One of DuckDB's powerful features is the ability to run SQL queries **directly on Python data structures** like Pandas DataFrames, Polars DataFrames, NumPy arrays, and Arrow tables, without requiring an import step. This is achieved via **replacement scans** and object registration.

Direct DataFrame Querying: If you have a Pandas DataFrame `df` in your Python session, you can query it with DuckDB by referring to it by name in `conn.sql()`. Example:

```
import pandas as pd
df = pd.DataFrame({"i":[1,2,3], "j":["a","b","c"]})
print(conn.sql("SELECT * FROM df").fetchall())  # query the Pandas DataFrame [57]
# Output: [(1, 'a'), (2, 'b'), (3, 'c')]
```

DuckDB will detect that `df` is a DataFrame in the local Python scope and read from it as if it were a table [58]. This is called a **replacement scan** – DuckDB replaces the table name in the SQL with a scan of the DataFrame's data.

Supported in-memory objects for replacement scans include: - Pandas DataFrame (`pandas.DataFrame`) - Polars DataFrame (`polars.DataFrame`) and LazyFrame (`polars.LazyFrame`) - PyArrow Table / RecordBatch / Dataset / Scanner - NumPy structured arrays or 2D arrays (as tables with one or more columns) - DuckDB relations (you can pass a `DuckDBPyRelation` as a table in another query) - Python lists (certain simple cases) and more [58].

Polars integration: Similarly, if you have a Polars DataFrame `pl_df`, you can do `duckdb.sql("SELECT * FROM pl_df")` to query it [59] [60]. DuckDB leverages Arrow as the interchange, so Polars data is scanned efficiently [25]. Polars LazyFrames can be queried the same way by name.

Arrow objects: If you use PyArrow to produce a Table or RecordBatch, you can query it directly. For example:

```
table = pyarrow.table({'x': [1,2,3]})
res = conn.sql("SELECT sum(x) FROM table").fetchall()  # treat Arrow table as
DuckDB table
```

This works because Arrow objects in scope are recognized by DuckDB's replacement scan mechanism [58].

NumPy: You can register a NumPy array as a table too. For instance, a 2D NumPy array can be seen as a table with one column (if structured) or multiple columns (if a plain 2D array). E.g. if `arr = np.array([[1,2],[3,4]])`, then `duckdb.sql("SELECT * FROM arr")` would yield the 2x2 array as two columns. (NumPy support may require arrays in structured dtype for proper column names; for complex cases, convert to Pandas or Arrow first.)

Explicit object registration: In cases where the object isn't a simple variable or is out of scope, you can explicitly register it:

```
duckdb.register('my_df_view', df)  # register DataFrame as table name
"my_df_view"
conn.sql("SELECT * FROM my_df_view").fetchall()
```

This creates a temporary view in the DuckDB connection named `my_df_view` that refers to the DataFrame [61]. You can register Arrow tables or Polars DataFrames similarly. Registration is also useful if you want to control name precedence or avoid confusion.

Name precedence: If a name could refer to multiple things (e.g. you have a DataFrame and a table in DuckDB with the same name): - A **registered** object (via `register`) takes highest precedence [62] [63]. - Then comes any DuckDB table/view with that name. - Lastly, DuckDB will try a replacement scan of a Python variable with that name [62] [63]. Thus, registered names let you override existing tables safely.

After querying a DataFrame/Arrow, you might want to persist the results or DataFrame itself as a DuckDB table: - **CREATE TABLE AS**: `conn.execute("CREATE TABLE mytable AS SELECT * FROM df")` will materialize the DataFrame `df` into a new DuckDB table on disk (or in-memory if the DB is in-memory) [64]. - **INSERT INTO**: Similarly, if `mytable` exists, `conn.execute("INSERT INTO mytable SELECT * FROM df")` appends DataFrame data to it [64]. - Under the hood, DuckDB will efficiently fetch the DataFrame's data in chunks. For Pandas, DuckDB analyzes object-dtype columns with a sample (default 1000 rows) to decide column types. If you have mixed types in `object` columns and see errors like "Failed to cast value", consider increasing the sample size with `SET pandas_analyze_sample=10000` (or higher) [65] [66].

Example: Combining Pandas and DuckDB:

```
# Assume conn is open
import pandas as pd
sales = pd.DataFrame({
    "region": ["US","EU","US"], "product": ["A","A","B"], "revenue": [100, 150,
200]
})
# Query with DuckDB:
result = conn.sql("""
    SELECT region, SUM(revenue) AS total_rev
    FROM sales
    GROUP BY region
""").to_df()
print(result)
# Output:   region  total_rev
#           0   EU        150
#           1   US        300
```

In this query, `sales` is a pandas DataFrame used as if it were a table. The result is collected into another DataFrame.

Parameterized Queries and Prepared Statements

DuckDB's Python client supports **parameterized SQL** queries, which is critical for both performance (when reusing queries) and security (avoiding SQL injection). The API follows DB-API conventions:

- Use `?` placeholders in your SQL for positional parameters [67] [68].
- Alternatively, use `$1, $2, ...` numbered placeholders, or `$name` for named parameters [69] [70].
- Supply the parameter values as a second argument to `execute()` (either a list/tuple for positional parameters, or a dict for named parameters).

Example – positional parameters:

```
# Create a table and insert using parameters
conn.execute("CREATE TABLE items(item VARCHAR, value INT, qty INT)")
# Single insert
conn.execute("INSERT INTO items VALUES (?, ?, ?)", ["laptop", 2000, 1])
【29†L585-L593】
# Batch insert multiple rows with executemany
conn.executemany("INSERT INTO items VALUES (?, ?, ?)", [
    ["chainsaw", 500, 10],
    ["iphone", 300, 2]
])【29†L589-L596】
```

```
# Query with a parameter
min_value = 400
conn.execute("SELECT item FROM items WHERE value > ?", [min_value])  (29†L595-
L599)
print(conn.fetchall())  # e.g., [('laptop',), ('chainsaw',)]
```

In the above: - The `?` placeholders are replaced by the values in order [67]. DuckDB prepares the query, then binds the parameters. - `executemany()` was used to insert multiple rows in one call [71] (it prepares the statement once and executes it for each parameter list).

Named parameters: Use the `$name` syntax and provide a dictionary:

```
result = duckdb.execute(
    "SELECT $greet || ', ' || $noun",
    {"greet": "Hello", "noun": "DuckDB"}
).fetchone()
print(result)  # -> ('Hello, DuckDB',)
```

Here `$greet` and `$noun` in the SQL are replaced by the corresponding dictionary values [72] [73]. Named parameters can be reused multiple times in the query (each occurrence of `$name` will use the same value).

DuckDB also accepts numbered parameters like `$1, $2` which correspond to the 1st, 2nd entries in the parameter list provided [69]. For example, `"SELECT $1, $1, $2"` with `["duck","goose"]` would reuse the first parameter in multiple places [70].

DB-API methods: After executing, use: - `fetchone()` – get next row of result (or `None` if no more). - `fetchall()` – get all rows as a list of tuples. - `fetchmany(n)` – get next *n* rows. - These can be called on the connection (since connection acts as cursor) or on the result of `execute()`. Also, `description` property gives column names of the last result [74].

Important: Avoid using Python string formatting to inject variables into SQL (which is vulnerable to injection attacks). Instead, always use placeholders as above [75] [76].

Note on performance: DuckDB can prepare statements under the hood for parameterized queries. If you call the same SQL with different parameters repeatedly via `execute()`, DuckDB will likely reuse the cached prepared statement, giving good performance. Using `executemany` is fine for a modest number of inserts, but for *very large* bulk inserts, it's recommended to use other techniques: - Create a pandas DataFrame and do `CREATE TABLE AS SELECT * FROM df` (bulk load via replacement scan, as discussed in the previous section). - Use the DuckDB **Appender** interface or the `COPY` command for massive inserts rather than thousands of `executemany` calls [77].

Defining Python UDFs (Scalar and Vectorized)

DuckDB allows you to define custom functions in Python and call them from SQL. These **Python UDFs** can run per row (scalar) or on columnar data (vectorized via Arrow). This enables extending DuckDB's SQL with arbitrary Python logic, while still leveraging DuckDB for data access.

Use `conn.create_function(name, function, parameters, return_type, **options)` to register a UDF [78] [79]: - **name**: SQL name for the function (string). - **function**: the actual Python callable (e.g. a `def` or `lambda`). - **parameters**: list of DuckDB data types for the input parameters. You can use types from `duckdb.sqltypes` (e.g. `INTEGER`, `VARCHAR`) or strings like `'INT'`. If the function has Python type annotations, you can often pass `None` to have DuckDB infer types [80] [81]. - **return_type**: DuckDB type of the return value. - **options** (optional): - `type='native'` (default) or `'arrow'`: whether to use **native scalar** mode (calls the function for each value) or **Arrow** mode (calls the function on a batch of values as Arrow arrays) [82]. Arrow mode is typically much faster for large data due to vectorized execution. - `null_handling`: `'NULL'` (default) or `'special'`. By default, if any argument to the UDF is NULL, DuckDB will **not call** your function and will automatically return NULL (NULL in, NULL out behavior) [83]. If you want to handle NULLs inside your function, set `null_handling='special'` – then your function will receive Python `None` for NULL inputs and you can decide what to return [83]. - `exception_handling`: `'throw'` (default) or `'return_null'`. By default, if your Python UDF raises an exception for any row, it will bubble up and abort the query [84]. If you prefer that errors in the UDF simply produce NULL in the result (and the query continues), use `exception_handling='return_null'` [84]. - `side_effects`: `False` (default) or `True`. Default assumes the function is pure (same output for same input). If your function has side effects or uses external state/randomness, set `side_effects=True` to avoid potential optimizations that assume determinism [85].

Example 1: Simple scalar UDF:

```
def add_ten(x: int) -> int:
    return x + 10

conn.create_function("add_ten", add_ten)  # DuckDB infers types from annotation
 [80] [81]
print(conn.sql("SELECT add_ten(5)").fetchone())  # (15,)
```

Because we annotated the Python function with types, we didn't explicitly pass `parameters` or `return_type`. DuckDB recognized the input as int and output as int automatically [80] [81].

Example 2: Handling NULLs and exceptions:

```
def safe_divide(a: float, b: float) -> float:
    # Return None (SQL NULL) if division by zero
    if b == 0:
        return None
    return a / b
```

```
conn.create_function("safe_divide", safe_divide, [duckdb.sqltypes.DOUBLE,
duckdb.sqltypes.DOUBLE], duckdb.sqltypes.DOUBLE,
                     null_handling='special', exception_handling='return_null')
# null_handling='special' passes NULLs to our function so we can handle (if b or
a is None).
# exception_handling='return_null' ensures an unexpected error would yield NULL
instead of stopping the query.
```

Now `SELECT safe_divide(10, 0)` will return SQL NULL (handled by our code), and any internal exception would also result in NULL rather than an error.

Vectorized (Arrow) UDFs: By setting `type='arrow'`, DuckDB will pass entire columns as Arrow arrays to your function, instead of one value at a time [82]. This allows your function to use vectorized libraries (NumPy, pandas, etc.) for speed. For example:

```
import numpy as np
def vector_add(x_arrow_array, y_arrow_array):
    # Convert Arrow arrays to numpy (zero-copy)
    x = np.array(x_arrow_array)
    y = np.array(y_arrow_array)
    return x + y  # elementwise addition

conn.create_function("vector_add", vector_add, [duckdb.sqltypes.BIGINT,
duckdb.sqltypes.BIGINT], duckdb.sqltypes.BIGINT,
                     type='arrow')
print(conn.sql("SELECT vector_add(i, j) FROM (VALUES (1,2), (3,4))
t(i,j)").fetchall())
# Output: [(3,), (7,)]
```

Here, DuckDB calls `vector_add` only twice (once per chunk of data) instead of once per row, and the function performs NumPy addition which is very fast on arrays. **Note:** The Arrow arrays `x_arrow_array` can be directly used or converted to pandas Series, etc. If using NumPy, ensure the data fits in memory.

Remember to **install pyarrow** in your environment if using Arrow UDFs. DuckDB's Python package comes with Arrow support, but the `pyarrow` library is needed to manipulate Arrow types in Python.

Resource management: Python UDFs execute within the DuckDB process (in the Python environment). Heavy use of UDFs can impact performance; whenever possible, see if DuckDB's built-in SQL functions can achieve the same result (as they are optimized in C++). However, UDFs are invaluable for custom logic.

To **remove** a UDF, use `conn.remove_function('function_name')` [86]. This will unregister the function from the connection's catalog.

DuckDB UDFs also support **partial application** and closures. You can bind extra arguments to your Python function using `functools.partial` before registering [87] [88]. This way, you can create parametrized UDFs (the partial arguments are set at create time). For example, you could bind a logging function or a constant to your UDF and only expose one parameter to SQL.

Using DuckDB Extensions (httpfs, JSON, Excel, etc.)

DuckDB's functionality can be extended with **extensions**—modular add-ons that provide new SQL functions or data sources. In Python, many extensions are included in the DuckDB package and can be loaded on demand. Key extensions for data engineering include **httpfs**, **json**, **excel**, **delta**, **iceberg**, etc., which cover connectivity and file formats.

Installing/Loading Extensions: DuckDB has an online extension repository. By default, if you use an extension feature, DuckDB will try to **auto-install and load** the extension (controlled by `autoinstall_known_extensions` and `autoload_known_extensions` settings, both true by default) [89]. For example, selecting from a `'s3://...` URL will auto-install the `httpfs` extension if not already loaded. However, you can also explicitly install/load:

```
conn.execute("INSTALL 'json';")   # one-time download of extension (if not
built-in) [89]
conn.execute("LOAD 'json';")      # load it into current session [90]
# Shorthand: LOAD will auto-install if needed.
```

For Python, many extensions (like `json` and `httpfs`) are **bundled** and may not require an explicit install. In fact, the `json` extension is *built-in* to DuckDB's standard builds and is loaded transparently on first use [89]. The code above is mainly needed if you want to ensure an extension is loaded upfront or if auto-loading is disabled.

httpfs: This extension enables DuckDB to access web URLs (HTTP/HTTPS) and cloud storage (S3, GCS, Azure Blob via HTTP signatures). It also brings support for faster encryption via OpenSSL. In practice, you might not need to manually load `httpfs` in the Python client, because DuckDB will handle it when you use an `http://` or `s3://` path. If you want to be sure:

```
conn.execute("LOAD httpfs;")  # ensure httpfs is loaded (requires internet if
not pre-packaged)
```

Once loaded, you can query remote files as described in File I/O section. `httpfs` also unlocks encryption features; for example, attaching an encrypted DuckDB file benefits from OpenSSL in `httpfs` for speed [91].

json: The JSON extension provides the `read_json` function and JSON data type support. In the Python client, this is usually already available without explicit loading [89]. It allows querying JSON files and using functions like `json_extract`. If you encounter a scenario where JSON functions aren't found, you might

`LOAD json;` explicitly. JSON extension introduces a `JSON` type and associated functions for querying JSON structures.

excel: The Excel extension allows DuckDB to read `.xlsx` files and provides an Excel-specific `text()` formatting function (for Excel's TEXT spec). To use:

```
conn.execute("INSTALL 'excel';")
conn.execute("LOAD 'excel';")
```

After loading, you can use `read_xlsx('file.xlsx', ...)` in SQL to import Excel sheets [45] (as discussed in File I/O). You can specify sheet names and ranges. DuckDB treats each worksheet as a table. Writing to Excel is done via `COPY ... TO 'file.xlsx' (FORMAT XLSX)`.

delta: The Delta Lake extension allows reading Delta Lake table directories (which consist of Parquet files plus a transaction log). Usage:

```
INSTALL 'delta';
LOAD 'delta';
SELECT * FROM delta.read_delta('path/to/delta_table');
```

This will read the Delta log and present a queryable table of the latest snapshot [50]. You can also specify a specific version or timestamp if needed (the extension provides parameters for that). The `delta` extension depends on **deltalake** under the hood; ensure your DuckDB python package includes it (the auto-install will fetch it if not present).

iceberg: The Iceberg extension allows DuckDB to query Apache Iceberg tables. Iceberg is a table format that can be cataloged in Hive Metastore, AWS Glue, Nessie, etc. After loading `iceberg` (`INSTALL 'iceberg'; LOAD 'iceberg';`), you typically need to configure a catalog:

```
SET iceberg_catalog='<name>';
SET <name>.<param> = '<value>';
```

(for example, setting a Glue or Hive catalog). Once configured, you can do `SELECT * FROM my_iceberg_table` directly. Alternatively, `iceberg` extension might allow a direct table function or attach. This extension is more involved and may require reading DuckDB docs for proper configuration of the catalog (AWS credentials, etc.). But once set up, it enables analytics on Iceberg datasets seamlessly.

Other extensions: DuckDB has many other extensions (for full list, see DuckDB docs): - *Postgres* and *MySQL* extensions: allow attaching external Postgres/MySQL databases and querying them via DuckDB (acting like a federated query engine). - *sqlite* extension: allows attaching a SQLite database file and querying it (as seen in the ATTACH examples) [92]. - *fts* (full-text search), *fts_index*: for full text indexing and search functions. - *geospatial* (if available): functions for GIS operations. - *motherduck* (if using DuckDB with MotherDuck cloud) etc.

Each extension typically has an **INSTALL** (one-time) and **LOAD** (per session) step, unless it's built-in. If you want DuckDB to load certain extensions on every connection, you can set the config option or use the environment variable `DUCKDB_INSTALL_ALL_EXTENSIONS=1` to pre-install all known ones.

Keep in mind that **community/third-party extensions** need to be allowed by config (`allow_unsigned_extensions` for unsigned ones). By default, official extensions are signed and safe to autoinstall. If running in a restricted environment, you might disable auto-install for security.

Working with Multiple Databases and Storage

DuckDB can attach multiple databases in one connection, enabling data transfer or cross-database queries. It also provides SQL commands to **export** an entire database and **import** it elsewhere.

ATTACH and Multiple Databases

The `ATTACH` statement adds another DuckDB database to your current connection's catalog [93]. This is similar to attaching additional files in SQLite. Each attached database gets an alias name and can be queried with that qualifier.

Example:

```
ATTACH 'sales.duckdb' AS sales_db;
ATTACH 'analytics.duckdb' AS analytics_db (READ_ONLY);
```

This attaches two database files, naming them `sales_db` and `analytics_db`. Now you can do cross-database queries:

```
SELECT s.customer_id, s.amount, a.segment
FROM sales_db.sales_table s
JOIN analytics_db.customer_segments a ON a.id = s.customer_id;
```

Here `sales_db.sales_table` refers to the table `sales_table` in the attached `sales_db` database. You can attach as many DuckDB files as needed. By default, attaching a file is read-write, but adding `(READ_ONLY)` makes it read-only [94] [95].

- **Inferring alias**: If you do `ATTACH 'data.duckdb';` without `AS name`, DuckDB uses the filename (`data`) as the alias [96].
- **Detaching**: Use `DETACH alias;` to detach the database [97]. After detach, its tables are no longer accessible in the main connection.
- **Listing databases**: `SHOW DATABASES;` will list the main and attached ones [98].
- **Switching default database**: `USE alias;` changes the default context to that attached DB for subsequent unqualified queries [99] (you can always still refer to others by prefix).

You can also attach **remote DuckDB files** via URLs:

```
ATTACH 's3://my-bucket/warehouse.duckdb' AS wh (READ_ONLY);
```

With `httpfs` loaded and proper credentials, DuckDB can attach a database stored on S3 or HTTP(s) (read-only by default) [100] [101].

Additionally, `ATTACH` supports attaching non-DuckDB databases via extensions:

```
LOAD sqlite;
ATTACH 'legacy.db' AS legacy (TYPE sqlite);
```

If the `sqlite` extension is loaded, this attaches a SQLite file and you can query it through DuckDB [102]. Similarly, `ATTACH ... (TYPE postgres)` or `(TYPE mysql)` can create links to those systems, if configured.

Data transfer between DBs: Attaching multiple DuckDB files allows transferring data: - You can copy a table from one to another with `CREATE TABLE newdb.mytable AS SELECT * FROM olddb.mytable;` [103]. - You can insert or select across attached DBs as shown in the join example above. - DuckDB 1.2+ introduced `COPY FROM DATABASE`:

```
COPY FROM DATABASE old_db_alias TO new_db_alias;
```

which can copy all tables (optionally filtered) between databases efficiently [104].

Encryption: DuckDB supports **transparent encryption** of database files. If an attached database is encrypted, you supply an encryption key:

```
ATTACH 'secret.duckdb' AS secret_db (ENCRYPTION_KEY 'mySecretPwd');
```

This will decrypt on the fly (requires `httpfs` for optimized AES) [105] [106]. Likewise, you can **create** encrypted databases by attaching with an `ENCRYPTION_KEY` and then copying data in.

Exporting and Importing a Database

When upgrading DuckDB versions or migrating data, you may use `EXPORT DATABASE` and `IMPORT DATABASE`:

- `EXPORT DATABASE 'dir_path' (FORMAT ...)` dumps the entire contents of the current database (schemas, tables, views, sequences) to the specified directory [107] [108]. By default, it exports as CSV files with a schema.sql to recreate schema and load.sql with COPY commands [108] [109]. You can choose `FORMAT parquet` to export tables as Parquet instead [110]. Options like `COMPRESSION zstd` and `ROW_GROUP_SIZE` apply to Parquet export [111]. Example:

```
EXPORT DATABASE 'backup_dir' (FORMAT parquet);
```

This creates `backup_dir/` containing `schema.sql`, `load.sql`, and all table data as Parquet files [108] [112].

- `IMPORT DATABASE 'dir_path'` does the reverse: it assumes the directory has an exported schema and data, and loads it into the current database [113] [107]. Internally this executes the `schema.sql` (creating tables) and then the `load.sql` (COPYing data in) [109] [114]. You can also run those files manually. For convenience, `PRAGMA import_database('dir_path')` is an equivalent call [115].

These commands are useful for migrating a database to a newer DuckDB version that isn't backward-compatible: you `EXPORT` in the old version, then `IMPORT` in the new version [116] [117]. They can also serve as a backup mechanism (producing human-readable schema and data files).

Performance Tuning and Monitoring

DuckDB is designed to be fast out-of-the-box, but you can tune its execution and monitor query performance using several options:

- **Parallel Threads**: DuckDB will use multiple threads for queries by default. The default thread count is typically the number of cores on your machine (for heavy queries). You can change it:

```
SET threads = 4;
```

This restricts DuckDB to 4 parallel threads [118] [119]. You can also set this at connection time via `duckdb.connect(config={'threads': 8})`. For I/O-bound queries, more threads might help up to a point; for CPU-bound, set according to hardware.

- **Memory Limit**: By default, DuckDB uses up to all available memory. You can impose a limit:

```
SET memory_limit = '2GB';
```

This would make DuckDB try not to exceed 2 GiB of memory [120] [121]. It's a soft limit; DuckDB will attempt to stay under it by spilling to disk if necessary. You can also configure this at connect via `config={'memory_limit': '10GB'}`.

- **Temporary Directory**: DuckDB uses disk for large operations that spill (e.g. sorting large data). By default it uses the OS temp directory. You can set `SET temp_directory = '/path/to/tmp'` or even `temp_directory = ''` to run purely in-memory (no spilling to disk).

- **Progress Bar**: For long-running queries in interactive use, DuckDB can show a progress bar. Enable it with:

```
SET enable_progress_bar = true;
```

Then, queries running over a certain duration will print a progress indicator to the console [122]. Note this is mainly effective in interactive Python/terminal usage; in Jupyter notebooks, the output might appear after completion.

- **Query Planning and Profiling**: Use `EXPLAIN` to see the planned execution:

```
EXPLAIN SELECT * FROM huge_table WHERE ...;
```

This will output the query plan with estimated row counts for each step [123]. No data is actually fetched. If you want to both run the query and get actual timing and row counts, use `EXPLAIN ANALYZE`:

```
EXPLAIN ANALYZE SELECT * FROM huge_table WHERE ...;
```

DuckDB will execute the query and then output the plan with actual runtime metrics (cumulative time per operator, actual rows processed, etc.) [124]. This is extremely useful for performance debugging.

- **Profiling details**: For programmatic access to profiles, you can enable profiling output as JSON:

```
SET enable_profiling = json;
SET profiling_mode = detailed;
```

and then after a query, fetch the profile via `SELECT * FROM duckdb_profiles();` or similar. DuckDB also has a GUI profile viewer (if you save the JSON). Refer to DuckDB's profiling docs for more on this.

- **Query optimization**: DuckDB generally optimizes automatically. But you can influence the planner with certain pragmas (e.g., force or disable index usage, join order, etc.), though such tweaks are rarely needed. One common performance tip: ensure proper **indexes** on large tables if doing point lookups. DuckDB supports explicit indexes (`CREATE INDEX ON table(col)`), but often a sequential scan is fine given its columnar nature.

- **Monitoring resource usage**: Functions like `duckdb_status()` and tables like `duckdb_tasks()` can show memory and thread usage if needed. For example, `SELECT * FROM duckdb_settings()` lists all configuration options and their current values [125] [126], which is useful to verify your settings.

In summary, for heavy workloads ensure you have appropriate `threads` and `memory_limit` set, use `EXPLAIN ANALYZE` to find bottlenecks, and consider `indexes` or re-partitioning data if queries are still

slow. DuckDB's default vectorized engine is usually very fast for scans and aggregations, so query design (e.g., avoiding extremely large joins or *ORDER BY* without limits on huge sets) is also important.

Secrets Management for Cloud Access

When working with cloud storage (like Amazon S3, Google Cloud, Azure) or other external services, DuckDB's **Secrets Manager** provides a secure way to manage credentials inside the database environment. Instead of embedding keys in file paths or code, you can create secrets and let DuckDB handle authentication for you.

Creating Secrets

Use the `CREATE SECRET` SQL statement to store credentials in DuckDB's Secrets Manager [127] [128]. For example, to store AWS S3 keys:

```
CREATE SECRET myaws (
    TYPE s3,
    KEY_ID 'AKIA****************',
    SECRET '********************************',
    REGION 'us-west-2'
);
```

This registers a temporary secret named `myaws` for AWS S3 with the given Access Key ID, Secret Key, and region [129]. By default, this secret is **temporary** (lives only for the duration of the DuckDB session) [130] [131]. Any S3 access will now use this credential when appropriate.

If you want the secret to persist across sessions (stored securely on disk), use `CREATE PERSISTENT SECRET name (...)` [132]. Persistent secrets are saved (on Linux/Mac in `~/.duckdb/stored_secrets`, with file permissions 600) and loaded next time you start DuckDB [133]. *Note:* Persistent secrets on local DuckDB are currently stored in **unencrypted** form on disk (protected by file permissions) [134], so treat the DuckDB config directory as sensitive. MotherDuck (cloud) stores secrets encrypted [135] [136].

DuckDB supports different **secret types** for different services [137]: - `s3` – Amazon S3 (also works for other S3-compatible storage like MinIO). - `azure` – Azure Blob Storage. - `gcs` – Google Cloud Storage. - `http` – HTTP basic auth (if needed for web URLs). - `huggingface` – HuggingFace Hub tokens. - etc. (Extensions can add more types: e.g., database secrets for mysql/postgres if those extensions are used) [138].

Each type has certain parameters. For S3/GCS/R2, you saw `KEY_ID`, `SECRET`, `REGION`, and optional `SESSION_TOKEN`, `URL_STYLE`, etc. [139] [140]. For Azure, you might provide an account name and key, etc. Check DuckDB docs for each secret type's parameters.

Alternatively, DuckDB can obtain credentials from your environment using **providers**:

```
CREATE SECRET myaws_auto (
    TYPE s3,
    PROVIDER credential_chain
);
```

This would tell DuckDB to use AWS's default credential chain (environment variables, IAM roles, or config files) to fetch credentials [141] [142]. The `credential_chain` provider exists for AWS, Azure, and others to seamlessly use your existing login (e.g., `aws configure` or Azure CLI auth).

Using Secrets for Cloud Access

Once a secret is created, DuckDB will automatically use it when accessing matching resource URLs. The **scope** of a secret determines for which paths it applies [143] [144]. By default, if you don't specify `SCOPE`, it often applies to all buckets for that service (or a default prefix). You can narrow it, e.g.:

```
CREATE SECRET myaws2 (
    TYPE s3,
    KEY_ID 'AKIA...', SECRET '...',
    REGION 'us-west-2',
    SCOPE 's3://my-bucket/data/'
);
```

This secret would only be used for S3 paths under `s3://my-bucket/data/` [145]. Scoping is important if you have multiple credentials (e.g., two different AWS accounts or different buckets with different keys) [146]. DuckDB will choose the secret with the longest matching prefix for a given path [147].

For public buckets or no-auth scenarios, you might not need a secret (DuckDB will attempt anonymous access). But for private data, define the secret then simply run your queries:

```
conn.execute("CREATE SECRET awsread (TYPE s3, KEY_ID '...', SECRET '...', REGION
'us-east-1')");
# Now query S3 parquet after secret is set:
conn.sql("SELECT COUNT(*) FROM 's3://my-bucket/data/
records.parquet'").fetchall()
```

The query will use the credentials stored in `awsread` to access the file. No need to embed keys in the path.

You can inspect which secret DuckDB is using for a path with:

```
SELECT * FROM which_secret('s3://my-bucket/data/records.parquet', 's3');
```

This table function returns the secret name and whether it's persistent, etc., that would be used for that path [148] [149].

To list all defined secrets (without revealing secrets themselves):

```
SELECT * FROM duckdb_secrets();
```

This will show secret names, types, and (for persistent) whether stored on disk or motherduck, etc., with sensitive values redacted [150] [151].

To delete a secret:

```
DROP SECRET awsread;
DROP PERSISTENT SECRET myaws;
```

This removes it from memory, and if persistent, from disk as well [152].

Security: By default, DuckDB will redact secret values in logs and when you query `duckdb_secrets()`. You can override (not recommended on shared systems) by setting `allow_unredacted_secrets=true` to see keys in plain text [150]. Also, a config `allow_persistent_secrets` controls if persistent secrets are allowed at all (default true) [153].

Using the Secrets Manager is the **recommended way** to handle cloud credentials because: - It avoids putting secrets in query text (which could be logged or seen by others). - It scopes the usage to only the intended operations. - In cloud DuckDB (MotherDuck), secrets are encrypted and managed securely [135] [136]. - You can manage and revoke secrets centrally (e.g., drop the secret to revoke access without changing code).

In summary: **create a secret once, then use DuckDB to read/write data freely**. This makes your Python and SQL code cleaner and safer.

Example Workflow: Combining Features

Finally, let's walk through a realistic scenario that uses multiple DuckDB features together. Suppose we have a pandas DataFrame of sales data that we want to analyze, save as Parquet, and then load into a persistent DuckDB database for future use.

```
import duckdb
import pandas as pd

# Sample in-memory data
sales_df = pd.DataFrame({
    "order_id": [1001, 1002, 1003],
```

```
    "product": ["A", "B", "A"],
    "amount": [500, 300, 450],
    "region": ["US", "EU", "US"]
})

conn = duckdb.connect(database=':memory:')  # use an in-memory DuckDB for
analysis

# 1. Query the DataFrame directly with DuckDB
top_region = conn.sql("""
    SELECT region, SUM(amount) AS total_sales
    FROM sales_df
    GROUP BY region
    ORDER BY total_sales DESC
    LIMIT 1
""").fetchone()  # DuckDB reads from sales_df [57]

print(f"Top region by sales: {top_region[0]} with ${top_region[1]} in sales.")
# (This uses replacement scan to query the pandas DataFrame)

# 2. Save detailed results to a Parquet file using DuckDB's relation API
result_rel = conn.sql("SELECT * FROM sales_df WHERE amount > 400")  # lazy
relation
result_rel.to_parquet("high_value_sales.parquet")  # materialize to Parquet file
[30] [154]

# 3. Attach a new DuckDB database file and import the Parquet data
conn.execute("ATTACH 'analytics.duckdb' AS analytics_db")  # attach new
persistent DB [155]
conn.execute("""
    CREATE TABLE analytics_db.high_value_sales AS
    SELECT *
    FROM read_parquet('high_value_sales.parquet')
""")  # Create table in attached DB from Parquet [156]

# Verify the data in the new attached database
rows = conn.execute("SELECT COUNT(*) FROM
analytics_db.high_value_sales").fetchone()
print(f"Rows in analytics_db.high_value_sales: {rows[0]}")

conn.execute("DETACH analytics_db")  # detach the persistent database [97]
conn.close()
```

Let's break down what happened:

- We created a Pandas DataFrame `sales_df` and directly ran a DuckDB SQL query on it to find the top sales region. DuckDB treated `sales_df` as a table without any prior loading step [57].

- We then formed a DuckDB relation for all sales over 400. This `result_rel` was not executed until we explicitly saved it. Calling `to_parquet("high_value_sales.parquet")` executed the SQL and wrote the output to a Parquet file [30] [154]. We did not pull the data into Python at all here – DuckDB streamed it directly to Parquet.
- Next, we used `ATTACH` to create a new DuckDB file (`analytics.duckdb`) and gave it alias `analytics_db`. This is an empty database initially.
- We populated the attached database by selecting from the Parquet file. The `CREATE TABLE ... AS SELECT * FROM read_parquet(...)` statement reads the Parquet file and creates a table in `analytics_db` [156]. Now the data lives in the persistent DuckDB.
- We confirmed the row count, then detached the `analytics_db` file. We now have a DuckDB database file on disk containing the `high_value_sales` table for future use (you could later do `duckdb.connect('analytics.duckdb')` and query it).
- Throughout, we never manually iterated over the DataFrame or managed intermediate CSVs. DuckDB handled data transfer efficiently (DataFrame -> DuckDB -> Parquet -> DuckDB).

This example demonstrated: - SQL on a Pandas DataFrame (replacement scans). - Lazy relational operations and writing Parquet. - Using DuckDB as an ETL tool to move data into a persistent store (via ATTACH and read_parquet). - The combination of these features can replace a lot of boilerplate data engineering code with concise SQL.

This concludes the advanced reference guide for DuckDB's Python client. With these tools, you can seamlessly query in-memory data, work with local and cloud files, extend DuckDB with Python logic, and manage data across multiple formats and locations – all with the convenience of SQL and the efficiency of DuckDB's engine. Happy querying!

Sources: DuckDB Official Documentation [11] [58] [68] [157] [30] (and others as cited in text).

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [67] [68] [69] [70] [71] [72] [73] [74] [77] Python DB API – DuckDB
https://duckdb.org/docs/stable/clients/python/dbapi

[11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [24] [27] [28] [29] [30] [31] [32] [33] [154] Relational API – DuckDB
https://duckdb.org/docs/stable/clients/python/relational_api

[21] [22] [23] [25] [26] [59] [60] Integration with Polars – DuckDB
https://duckdb.org/docs/stable/guides/python/polars

[34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [51] [57] [58] [61] [62] [63] [64] [65] [66] Data Ingestion – DuckDB
https://duckdb.org/docs/stable/clients/python/data_ingestion

[45] [46] [47] [48] [156] Excel Import – DuckDB
https://duckdb.org/docs/stable/guides/file_formats/excel_import

[49] Excel Export - DuckDB
https://duckdb.org/docs/stable/guides/file_formats/excel_export.html

[50] Does DuckDB support Delta Lake? | Orchestra
https://www.getorchestra.io/guides/does-duckdb-support-delta-lake

52 53 54 118 119 121 122 125 126 153 Configuration – DuckDB
https://duckdb.org/docs/stable/configuration/overview

55 56 Using fsspec Filesystems – DuckDB
https://duckdb.org/docs/stable/guides/python/filesystems

75 76 120 150 Securing DuckDB – DuckDB
https://duckdb.org/docs/stable/operations_manual/securing_duckdb/overview

78 79 80 81 82 83 84 85 86 87 88 157 Python Function API – DuckDB
https://duckdb.org/docs/stable/clients/python/function

89 90 Installing and Loading the JSON extension – DuckDB
https://duckdb.org/docs/stable/data/json/installing_and_loading

91 92 93 94 95 96 97 98 99 100 101 102 103 105 106 155 ATTACH and DETACH Statements – DuckDB
https://duckdb.org/docs/stable/sql/statements/attach

104 116 117 Storage Versions and Format – DuckDB
https://duckdb.org/docs/stable/internals/storage

107 108 109 110 111 112 113 114 115 EXPORT and IMPORT DATABASE Statements – DuckDB
https://duckdb.org/docs/stable/sql/statements/export

123 124 Profiling Queries – DuckDB
https://duckdb.org/docs/stable/sql/statements/profiling

127 128 129 130 131 132 133 134 137 138 143 144 145 147 151 152 Secrets Manager – DuckDB
https://duckdb.org/docs/stable/configuration/secrets_manager

135 136 139 140 141 146 148 149 CREATE SECRET | MotherDuck Docs
https://motherduck.com/docs/sql-reference/motherduck-sql-reference/create-secret/

142 S3 API Support - DuckDB
https://duckdb.org/docs/stable/core_extensions/httpfs/s3api.html