Tabular data: CSV, Parquet and SQL with DuckDB

A huge amount of biological data ends up as a **table**: rows are things (genes, samples, species, BLAST hits) and columns are measurements or properties of those things. This lecture covers:

1. The file formats tables are stored in - **CSV/TSV** (text) and **Parquet** (binary, columnar)
2. **SQL**, a language for asking questions of tables
3. **DuckDB**, a small program (and Python package) that runs SQL directly on CSV and Parquet files
4. Using it on a real bioinformatics output: a BLAST table

Why tables, and why not Excel?

Spreadsheets are great for looking at a few hundred rows. They are a poor fit for bioinformatics data because:

- Excel stops at 1,048,576 rows. A single RNASeq count matrix, VCF, or BLAST result can be much bigger.
- You can't run Excel on the cluster, and pointing and clicking is not **reproducible** - there is no record of what you did.
- Excel "helpfully" changes your data. Gene names like `SEPT2` and `MARCH1` were silently converted to dates so often that the human gene nomenclature committee renamed the genes (now `SEPTIN2` and `MARCHF1`). Long IDs lose leading zeros or get turned into `1.2E+10`.

Instead we keep tables in plain files and process them with code - UNIX tools, Python, R, or SQL.

CSV and TSV files

A **CSV** (comma-separated values) file is plain text: one line per row, with columns separated by a **delimiter**. A **TSV** file uses a tab (`\t`) as the delimiter instead of a comma. The first line is usually a **header row** giving the column names.

Get the data used in this lecture (you used the first one in HW1):

```
mkdir -p duckdb_lecture
cd duckdb_lecture
curl -LO https://github.com/biodataprog/GEN220_data/raw/main/tabular/threatened-species.csv.gz
curl -LO https://github.com/biodataprog/GEN220_data/raw/main/tabular/airport-codes.csv.gz
zcat airport-codes.csv.gz | head -3
```

```
ident,type,name,elevation_ft,continent,iso_country,iso_region,municipality,gps_code,iata_code,local_code,
00A,heliport,Total Rf Heliport,11,NA,US,US-PA,Bensalem,00A,,00A,"40.07080078125, -74.93360137939453"
00AA,small_airport,Aero B Ranch Airport,3435,NA,US,US-KS,Leoti,00AA,,00AA,"38.704022, -101.473911"
```

(On a Mac use `zcat < file.gz` or `gzcat`.)

Quoting

What if a value itself contains a comma? The CSV convention is to put that field in **double quotes**. The `coordinates` column above is `"40.07080078125, -74.93360137939453"` - one field with a comma inside. Here is a heliport in Fort Collins whose *name* has a comma in it:

```
zcat airport-codes.csv.gz | grep '^2CO0,'
```

```
2CO0,heliport,"Heli-One American Support, LLC Heliport",4935,NA,US,US-CO,Fort Collins,2CO0,,2CO0,"40.5835
```

`cut` knows nothing about quotes - it splits on *every* comma. Ask for column 3 (name) and column 8 (municipality):

```
zcat airport-codes.csv.gz | grep '^2CO0,' | cut -d, -f3,8
```

```
"Heli-One American Support,US-CO
```

We got half of the name and the wrong column (`US-CO` is really column 7) because the extra comma shifted everything over by one. `cut -d,` and `awk -F,` are fine for simple files you made yourself, but for CSV files from other people use a tool that understands quoting: Python's `csv` module, pandas, R, or DuckDB. For example:

```
import csv, gzip
with gzip.open("airport-codes.csv.gz", "rt") as fh:
    for row in csv.reader(fh):
        if row[0] == "2C00":
            print(row[2], "|", row[7], "|", row[11])
```

```
Heli-One American Support, LLC Heliport | Fort Collins | 40.583585, -106.985331
```

TSV files avoid most of this problem, because tabs rarely appear inside values. That is one reason most bioinformatics tools (BLAST, GFF, BED, VCF, samtools) write tab-delimited output.

Missing values

A missing value is usually just an empty field - two delimiters in a row (,,). The `threatened-species` file has many:

```
zcat threatened-species.csv.gz | head -3
```

```
taxonid,kingdom_name,phylum_name,class_name,order_name,family_name,genus_name,scientific_name,taxonomic_a
31583,PLANTAE,TRACHEOPHYTA,MAGNOLIOPSIDA,MYRTALES,MYRTACEAE,Eugenia,Eugenia oreophila,Ridley,,,,LR/lc,
31584,PLANTAE,TRACHEOPHYTA,MAGNOLIOPSIDA,MYRTALES,MYRTACEAE,Eugenia,Eugenia orites,Ridley,,,,LR/cd,
```

Other files use `NA`, `NaN`, `.` or `-` for missing. Be careful - this can bite you. In the airport file `NA` means *North America*, but pandas treats the text `NA` as missing by default, so North America disappears:

```
import pandas as pd
d = pd.read_csv("airport-codes.csv.gz")
print(d["continent"].value_counts(dropna=False))
```

```
continent
NaN    36636
AS     11023
SA     10391
EU      9878
AF      3972
OC      3904
AN        44
Name: count, dtype: int64
```

(`pd.read_csv(..., keep_default_na=False)` fixes it.)

Types are not stored

A CSV file is only text. Nothing in the file says that `elevation_ft` is a number, that `taxonid` is an integer, or that `00A` should stay text. Every program that reads the file has to **guess** the type of each column (or you have to tell it). That guessing is slow for big files, and sometimes wrong.

Compressed CSV

CSV files compress very well with `gzip`. You don't need to uncompress them first - DuckDB, pandas, R and Python's `gzip` module all read `.csv.gz` directly.

Parquet

Parquet is a binary file format for tables, widely used in data science. Compared to CSV it is:

- **Columnar** - the values of each column are stored together. A query that only needs 2 of 14 columns only reads those 2 columns from disk.
- **Typed** - the file records that a column is an integer, a float, a string, a date, etc. No guessing when you read it.

- **Compressed** - each column is compressed separately, which works very well (a column of `CR`, `EN`, `VU`... is very repetitive).

You can't `cat` or `head` a Parquet file - you need a program that reads it (DuckDB, pandas/pyarrow, R `arrow`, polars). Here is how to convert our CSV to Parquet with DuckDB (introduced below), and compare sizes. Also make an uncompressed copy of the CSV for comparison:

```
duckdb -c "COPY (SELECT * FROM 'threatened-species.csv.gz')
  TO 'threatened-species.parquet' (FORMAT parquet, COMPRESSION zstd);"
gunzip -k threatened-species.csv.gz
ls -lh threatened-species.*
```

```
-rw-r--r--  1 jstajich  wheel    17M Sep 24 00:05 threatened-species.csv
-rw-r--r--  1 jstajich  wheel   3.5M Sep 24 00:05 threatened-species.csv.gz
-rw-r--r--  1 jstajich  wheel   3.1M Sep 24 00:09 threatened-species.parquet
```

The Parquet file is smaller than even the gzipped CSV. (Without `COMPRESSION zstd` DuckDB uses the faster but less compact `snappy` compression; that file was 5.0M.)

Now run the same query - count species in each IUCN category - on each file, with `.timer on` to print how long it took:

```
.timer on
SELECT category, count(*) AS n FROM 'threatened-species.csv' GROUP BY category ORDER BY n DESC LIMIT 3;
SELECT category, count(*) AS n FROM 'threatened-species.csv.gz' GROUP BY category ORDER BY n DESC LIMIT 3
SELECT category, count(*) AS n FROM 'threatened-species.parquet' GROUP BY category ORDER BY n DESC LIMIT
```

```
| category |   n   |
|----------|------:|
| LC       | 75247 |
| DD       | 19897 |
| EN       | 15506 |
Run Time (s): real 0.217 user 0.233415 sys 0.022297
| category |   n   |
|----------|------:|
| LC       | 75247 |
| DD       | 19897 |
| EN       | 15506 |
Run Time (s): real 0.425 user 0.397272 sys 0.017319
| category |   n   |
|----------|------:|
| LC       | 75247 |
| DD       | 19897 |
| EN       | 15506 |
Run Time (s): real 0.009 user 0.004155 sys 0.003138
```

Same answer, but the Parquet query took 0.009 seconds versus 0.2 s for the CSV and 0.4 s for the gzipped CSV (which has to be uncompressed as it is read) - about 25-50x faster. This file has only 143,732 rows; for tables with tens of millions of rows the difference is minutes versus seconds. A good habit: if you will query a big CSV more than once, convert it to Parquet first.

DuckDB

DuckDB is an **embedded analytical SQL database**. That sounds complicated, but in practice:

- It is a **single program** (`duckdb`) - no server to set up, no accounts, no administrator.
- It is also a **Python package** (`import duckdb`) and an R package.
- It can run SQL **directly on files**: CSV, TSV, `.csv.gz`, Parquet, JSON - no loading step needed.
- It is fast and uses all the CPUs you give it, and it can work on data bigger than memory.
- If you want, it can also store tables in a database file (`mydata.duckdb`).

(You may have heard of SQLite, which is similar but designed for many small updates, like an app's storage. DuckDB is designed for analysis: scanning, filtering and summarizing big tables.)

Getting DuckDB

Python package - in a virtual environment or conda environment:

```
pip install duckdb
python -c "import duckdb; print(duckdb.__version__)"
```

```
1.5.5
```

Command-line program - download the binary for your system from the DuckDB releases page. On the HPCC cluster (Linux, x86_64):

```
mkdir -p ~/bin
cd ~/bin
curl -LO https://github.com/duckdb/duckdb/releases/download/v1.5.5/duckdb_cli-linux-amd64.zip
unzip duckdb_cli-linux-amd64.zip
rm duckdb_cli-linux-amd64.zip
~/bin/duckdb --version
```

```
v1.5.5 (Variegata) d8cdaa33fd
```

(On a Mac use `duckdb_cli-osx-universal.zip`, or `brew install duckdb`.) Make sure `~/bin` is in your `PATH` (see the UNIX lectures) so you can just type `duckdb`.

On the cluster, also check whether a module is available - the list of installed software changes over time:

```
module avail duckdb
```

Running the DuckDB CLI

Type `duckdb` to get an interactive prompt (`D`); type SQL ending in `;` and press return. `.quit` (or Control-D) exits. You can also run a single command with `-c`, or a whole file of SQL:

```
duckdb                              # interactive, data kept in memory only
duckdb species.duckdb               # interactive, tables saved in the file species.duckdb
duckdb -c "SELECT 42 AS answer;"    # run one command and exit
duckdb < myqueries.sql              # run all the commands in a file
```

Commands that start with a `.` are settings for the CLI, not SQL. The most useful is `.mode`, which sets how results are printed. The default draws boxes around the results; the output in this lecture uses `.mode markdown`. Others are `.mode csv`, `.mode tabs`, `.mode line` (one value per line, good for wide tables) and `.mode table`. You can put `.mode markdown` in a file called `~/.duckdbrc` to make it your default, or run `duckdb -markdown`.

SQL basics

SQL (Structured Query Language, often said "sequel") has been the standard way to ask questions of tables for 50 years. It is used by nearly every database (DuckDB, SQLite, PostgreSQL, MySQL, BigQuery…) so it is worth learning the basics. A query describes *what* you want and the database figures out *how* to get it.

SQL keywords are not case sensitive (`select` works the same as `SELECT`), but writing them in capitals makes queries easier to read. Text values go in single quotes `'CR'`. A query can be spread across many lines and ends with `;`.

The examples below are all run in the `duckdb` CLI from the folder with the data files.

SELECT … FROM … LIMIT

`SELECT` picks the columns, `FROM` says which table - in DuckDB this can be a file name in single quotes. `LIMIT` prints only the first few rows. `SELECT *` means all columns.

```
SELECT genus_name, scientific_name, category
FROM 'threatened-species.csv.gz'
LIMIT 5;
```

```
| genus_name |   scientific_name   | category |
|------------|---------------------|----------|
| Eugenia    | Eugenia oreophila   | LR/lc    |
| Eugenia    | Eugenia orites      | LR/cd    |
| Eugenia    | Eugenia pahangensis | LR/cd    |
| Eugenia    | Eugenia pallidula   | VU       |
| Eugenia    | Eugenia pearsoniana | LR/cd    |
```

DuckDB figured out the delimiter, the header and the gzip compression by itself.

DESCRIBE

What columns are there, and what type did DuckDB guess for each? DuckDB reads a sample of the file and picks a type for each column. `taxonid` looked like whole numbers, so it became `BIGINT` (a 64-bit integer); everything else is text, called `VARCHAR`.

```
DESCRIBE SELECT * FROM 'threatened-species.csv.gz';
```

```
|     column_name     | column_type | null | key  | default | extra |
|---------------------|-------------|------|------|---------|-------|
| taxonid             | BIGINT      | YES  | NULL | NULL    | NULL  |
| kingdom_name        | VARCHAR     | YES  | NULL | NULL    | NULL  |
| phylum_name         | VARCHAR     | YES  | NULL | NULL    | NULL  |
| class_name          | VARCHAR     | YES  | NULL | NULL    | NULL  |
| order_name          | VARCHAR     | YES  | NULL | NULL    | NULL  |
| family_name         | VARCHAR     | YES  | NULL | NULL    | NULL  |
| genus_name          | VARCHAR     | YES  | NULL | NULL    | NULL  |
| scientific_name     | VARCHAR     | YES  | NULL | NULL    | NULL  |
| taxonomic_authority | VARCHAR     | YES  | NULL | NULL    | NULL  |
| infra_rank          | VARCHAR     | YES  | NULL | NULL    | NULL  |
| infra_name          | VARCHAR     | YES  | NULL | NULL    | NULL  |
| population          | VARCHAR     | YES  | NULL | NULL    | NULL  |
| category            | VARCHAR     | YES  | NULL | NULL    | NULL  |
| main_common_name    | VARCHAR     | YES  | NULL | NULL    | NULL  |
```

In SQL a missing value is called `NULL`. Empty fields in the CSV become `NULL`.

WHERE - filtering rows

`WHERE` keeps only the rows that match a condition. Combine conditions with `AND`, `OR` and `NOT`. Comparisons are `=`, `<>` (not equal), `<`, `>`, `<=`, `>=`.

```
SELECT scientific_name, main_common_name, category
FROM 'threatened-species.csv.gz'
WHERE class_name = 'MAMMALIA' AND category = 'CR'
LIMIT 5;
```

```
|           scientific_name            |       main_common_name        | category |
|--------------------------------------|-------------------------------|----------|
| Diceros bicornis ssp. michaeli       | Eastern Black Rhino           | CR       |
| Diceros bicornis ssp. minor          | South-eastern Black Rhino     | CR       |
| Cephalorhynchus hectori ssp. maui    | North Island Hector's Dolphin | CR       |
| Orcaella brevirostris                | Irrawaddy Dolphin             | CR       |
| Loris tardigradus ssp. nycticeboides | Highland Slender Loris        | CR       |
```

Other useful conditions:

- `category IN ('CR', 'EN', 'VU')` - matches any value in the list
- `main_common_name LIKE '%Salamander%'` - pattern match, `%` means "anything" (like `*` in the shell). `ILIKE` ignores upper/lower case.
- `main_common_name IS NULL` / `IS NOT NULL` - test for missing values (`= NULL` does **not** work)

ORDER BY - sorting

`ORDER BY` sorts the results by one or more columns; add `DESC` for largest first.

```
SELECT scientific_name, main_common_name
FROM 'threatened-species.csv.gz'
WHERE main_common_name LIKE '%Salamander%'
ORDER BY scientific_name
LIMIT 5;
```

```
|     scientific_name     |         main_common_name         |
|-------------------------|----------------------------------|
| Ambystoma amblycephalum | Blunthead Salamander             |
| Ambystoma andersoni     | Anderson's Salamander            |
| Ambystoma annulatum     | Ringed Salamander                |
| Ambystoma barbouri      | Streamside Salamander            |
| Ambystoma bishopi       | Reticulated Flatwoods Salamander |
```

COUNT(*) and AS

`count(*)` counts rows. `AS` gives a result column a name (an **alias**) - otherwise DuckDB invents one like `count_star()`.

```
SELECT count(*) AS n_species FROM 'threatened-species.csv.gz';
SELECT count(*) AS n_critical
FROM 'threatened-species.csv.gz'
WHERE category = 'CR';
```

```
| n_species |
|----------:|
| 143732    |
| n_critical |
|-----------:|
| 8576       |
```

DISTINCT

`DISTINCT` removes duplicate rows from the result - like `sort -u` in UNIX. `count(DISTINCT column)` counts the number of different values.

```
SELECT DISTINCT kingdom_name FROM 'threatened-species.csv.gz' ORDER BY kingdom_name;
SELECT count(DISTINCT family_name) AS n_families FROM 'threatened-species.csv.gz';
```

```
| kingdom_name |
|--------------|
| ANIMALIA     |
| CHROMISTA    |
| FUNGI        |
| PLANTAE      |
| n_families |
|-----------:|
| 2316       |
```

GROUP BY - counting by category

This is where SQL gets powerful. `GROUP BY` splits the rows into groups that share the same value, and then aggregate functions like `count(*)` are computed once *per group*. This is the SQL version of the UNIX `cut -f13 | sort | uniq -c` pipeline, or a Python dictionary of counters.

```
SELECT category, count(*) AS n
FROM 'threatened-species.csv.gz'
GROUP BY category
ORDER BY n DESC;
```

category	n
LC	75247
DD	19897
EN	15506
VU	15267
CR	8576
NT	7781
EX	455
LR/nt	443
LR/lc	387
LR/cd	120
EW	53

You can group by more than one column, e.g. `GROUP BY kingdom_name, category`. Every column in the `SELECT` must either be in the `GROUP BY` or inside an aggregate function.

Aggregates: AVG, MIN, MAX, SUM

Besides `count`, the common aggregate functions are `avg`, `min`, `max`, `sum`, and `median`. The species table has no numeric columns to summarize, so let's use the airports: elevation of large airports on each continent.

```
SELECT continent,
       count(*)                   AS n_airports,
       round(avg(elevation_ft))   AS mean_elev_ft,
       min(elevation_ft)          AS min_elev_ft,
       max(elevation_ft)          AS max_elev_ft
FROM 'airport-codes.csv.gz'
WHERE type = 'large_airport'
GROUP BY continent
ORDER BY mean_elev_ft DESC;
```

continent	n_airports	mean_elev_ft	min_elev_ft	max_elev_ft
SA	26	1661.0	16	10860
AF	48	1298.0	9	7630
NA	117	858.0	4	8466
AS	142	724.0	5	6903
EU	115	397.0	-11	1998
OC	16	88.0	5	434

Notice DuckDB kept `NA` as the text "NA" (North America), and it read the quoted fields correctly. Aggregate functions skip `NULL` values. Which airport is the highest?

```
SELECT ident, iata_code, municipality, iso_country, elevation_ft
FROM 'airport-codes.csv.gz'
WHERE type = 'large_airport'
```

```
ORDER BY elevation_ft DESC
LIMIT 3;
```

ident	iata_code	municipality	iso_country	elevation_ft
SPZO	CUZ	Cusco	PE	10860
MMTO	TLC	Ciudad de Toluca	MX	8466
SKBO	BOG	Bogota	CO	8361

HAVING - filtering groups

`WHERE` filters rows *before* grouping. `HAVING` filters groups *after* the aggregate is computed. Which classes have at least 1000 species listed as threatened (CR, EN or VU)?

```
SELECT class_name, count(*) AS n
FROM 'threatened-species.csv.gz'
WHERE category IN ('CR', 'EN', 'VU')
GROUP BY class_name
HAVING count(*) >= 1000
ORDER BY n DESC;
```

class_name	n
MAGNOLIOPSIDA	20851
LILIOPSIDA	3493
ACTINOPTERYGII	2630
AMPHIBIA	2471
INSECTA	1940
REPTILIA	1618
GASTROPODA	1440
AVES	1400

The order the parts of a query must be written in is: `SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY ... LIMIT`.

JOIN - combining two tables

The `category` column only has codes. Let's make a small second table that explains each code:

```
cat > iucn_categories.csv <<EOF
code,description,threatened
EX,Extinct,no
EW,Extinct in the Wild,no
CR,Critically Endangered,yes
EN,Endangered,yes
VU,Vulnerable,yes
NT,Near Threatened,no
LC,Least Concern,no
DD,Data Deficient,no
EOF
```

A `JOIN` matches up rows from two tables using a shared value - here `category` in the species table and `code` in the new table. We give each table a short alias (`s` and `c`) so we can say which table a column comes from.

```
SELECT s.scientific_name, s.category, c.description
FROM 'threatened-species.csv.gz' AS s
JOIN 'iucn_categories.csv' AS c ON s.category = c.code
WHERE s.genus_name = 'Gorilla';
```

```
|        scientific_name         | category |      description      |
|--------------------------------|----------|-----------------------|
| Gorilla beringei               | CR       | Critically Endangered |
| Gorilla beringei ssp. graueri  | CR       | Critically Endangered |
| Gorilla gorilla ssp. diehli    | CR       | Critically Endangered |
| Gorilla beringei ssp. beringei | EN       | Endangered            |
```

Once joined, you can group by columns from either table:

```
SELECT c.threatened, count(*) AS n
FROM 'threatened-species.csv.gz' AS s
JOIN 'iucn_categories.csv' AS c ON s.category = c.code
GROUP BY c.threatened;
```

```
| threatened |   n    |
|------------|-------:|
| false      | 103433 |
| true       | 39349  |
```

(DuckDB guessed that a column of only `yes`/`no` is a `BOOLEAN` true/false column.) These add up to 142,782, not 143,732. A plain `JOIN` (also called an `INNER JOIN`) drops rows that have no match in the other table. A `LEFT JOIN` keeps every row of the first table and fills in `NULL` where there is no match, which lets us find the rows that went missing:

```
SELECT s.category, count(*) AS n
FROM 'threatened-species.csv.gz' AS s
LEFT JOIN 'iucn_categories.csv' AS c ON s.category = c.code
WHERE c.code IS NULL
GROUP BY s.category;
```

```
| category |  n  |
|----------|----:|
| LR/cd    | 120 |
| LR/nt    | 443 |
| LR/lc    | 387 |
```

These are the old "Lower Risk" categories that we left out of our lookup table. Joins are how you would, for example, attach gene descriptions to a table of BLAST hits or DESeq2 results.

CREATE TABLE AS - saving results in a database

So far every query re-read the file. You can load a file (or the result of any query) into a **table** with `CREATE TABLE name AS SELECT ...`. If you started DuckDB with a database file (`duckdb species.duckdb`) the tables are saved in that file and are there the next time you open it; otherwise they disappear when you quit.

```
duckdb species.duckdb
```

```
CREATE TABLE species AS
    SELECT * FROM 'threatened-species.csv.gz';
CREATE TABLE amphibians AS
    SELECT scientific_name, order_name, family_name, category
    FROM species
    WHERE class_name = 'AMPHIBIA';
SHOW TABLES;
SELECT order_name, count(*) AS n FROM amphibians GROUP BY order_name ORDER BY n DESC;
```

```
|    name    |
|------------|
| amphibians |
| species    |
```

```
| order_name  |  n   |
|-------------|-----:|
| ANURA       | 6407 |
| CAUDATA     | 660  |
| GYMNOPHIONA | 191  |
```

Frogs (Anura), salamanders (Caudata) and caecilians (Gymnophiona). Use `DROP TABLE amphibians;` to delete a table.

COPY ... TO - writing files

`COPY` writes a table, or the result of a query in parentheses, to a file. This is how we made the Parquet file earlier.

```
COPY amphibians TO 'amphibians.parquet' (FORMAT parquet);
COPY (SELECT * FROM amphibians WHERE category = 'CR')
    TO 'amphibians_CR.csv' (HEADER, DELIMITER ',');
```

```
head -3 amphibians_CR.csv
```

```
scientific_name,order_name,family_name,category
Litoria booroolongensis,ANURA,PELODRYADIDAE,CR
Cophixalus mcdonaldi,ANURA,MICROHYLIDAE,CR
```

Use `DELIMITER '\t'` to write a TSV file.

SUMMARIZE - a quick look at every column

`SUMMARIZE` computes summary statistics for every column: min, max, approximate number of unique values, mean, standard deviation, quartiles, and the percentage of missing values. It's a great first thing to run on a new table. The full output is wide, so here we pick a few of its columns:

```
SELECT column_name, column_type, min, max, approx_unique, null_percentage
FROM (SUMMARIZE SELECT type, elevation_ft, continent, iata_code FROM 'airport-codes.csv.gz');
```

```
| column_name  | column_type |     min     |      max      | approx_unique | null_percentage |
|--------------|-------------|-------------|---------------|--------------:|----------------:|
| type         | VARCHAR     | balloonport | small_airport | 7             | 0.00            |
| elevation_ft | BIGINT      | -1266       | 17372         | 6757          | 18.91           |
| continent    | VARCHAR     | AF          | SA            | 7             | 0.00            |
| iata_code    | VARCHAR     | AAA         | ZZV           | 9675          | 88.29           |
```

19% of airports have no elevation and 88% have no IATA (3-letter) code. Try `SUMMARIZE 'airport-codes.csv.gz';` to see all of it.

DuckDB from Python

The `duckdb` Python package runs exactly the same SQL. `duckdb.sql(...)` runs a query; then get the results with:

- `.fetchall()` - a list of tuples, one per row
- `.fetchone()` - just the first row
- `.show()` - print a table
- `.df()` - a pandas data frame (needs pandas installed)

```
#!/usr/bin/env python3
import duckdb

# run a query and get the results back as a list of tuples
rows = duckdb.sql("""
    SELECT class_name, count(*) AS n
    FROM 'threatened-species.parquet'
```

```
    WHERE category = 'CR'
    GROUP BY class_name
    ORDER BY n DESC
    LIMIT 5
""").fetchall()

print(rows)
for class_name, n in rows:
    print(class_name, n, sep="\t")
```

```
[('MAGNOLIOPSIDA', 4314), ('LILIOPSIDA', 859), ('AMPHIBIA', 681), ('ACTINOPTERYGII', 535), ('GASTROPODA',
MAGNOLIOPSIDA   4314
LILIOPSIDA  859
AMPHIBIA    681
ACTINOPTERYGII  535
GASTROPODA  421
```

When a value in the query comes from a Python variable, don't paste it into the SQL string yourself - put a `?` in the query and pass the values in a list. This handles quoting for you (think of a name like `Hector's Dolphin`). Use `duckdb.connect()` to get a connection; `duckdb.connect("species.duckdb")` opens the database file we made above.

```
#!/usr/bin/env python3
import duckdb

con = duckdb.connect()   # in-memory database

for cls in ["AVES", "MAMMALIA", "AMPHIBIA"]:
    n = con.execute("""
        SELECT count(*) FROM 'threatened-species.parquet'
        WHERE class_name = ? AND category = 'CR'""", [cls]).fetchone()[0]
    print(cls, n)
```

```
AVES 233
MAMMALIA 152
AMPHIBIA 681
```

If you use pandas, `.df()` converts a result to a data frame, and DuckDB can even run SQL on a pandas data frame just by using its variable name in the `FROM`:

```
#!/usr/bin/env python3
import duckdb

# .df() turns the result into a pandas data frame
df = duckdb.sql("SELECT scientific_name, family_name, category "
                "FROM 'amphibians.parquet'").df()
print(df.shape)
print(df.head(3))

# DuckDB can also query a pandas data frame by its variable name
duckdb.sql("SELECT category, count(*) AS n FROM df "
           "GROUP BY category ORDER BY n DESC LIMIT 4").show()
```

```
(7258, 3)
           scientific_name     family_name category
0         Eurycea rathbuni  PLETHODONTIDAE       VU
1          Eurycea robusta  PLETHODONTIDAE       DD
2  Onychodactylus fischeri      HYNOBIIDAE       LC
+----------+-------+
```

```
| category |   n   |
| varchar  | int64 |
+----------+-------+
| LC       |  3252 |
| DD       |  1124 |
| EN       |  1087 |
| VU       |   703 |
+----------+-------+
```

(`.show()` draws the box with line-drawing characters; they are shown here as `+-|`.)

Example: querying BLAST results with SQL

BLAST's tabular output (`-outfmt 6`) is a TSV file with 12 columns and **no header row**. It's a perfect fit for SQL: "how many genes have a hit?", "what is the best hit for each gene?", "which hits are above 80% identity?" are all one query.

Make a BLAST table

We'll compare the first 200 *Saccharomyces cerevisiae* ORFs to the coding sequences (CDS) of the related yeast *Candida glabrata* (*Nakaseomyces glabratus*). On the cluster, load BLAST with `module load ncbi-blast` (check `module avail ncbi-blast` for the current name).

```
curl -LO https://github.com/biodataprog/GEN220_data/raw/main/genome/S_cerevisiae.ORFs.fasta.gz
curl -LO https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/002/545/GCF_000002545.3_ASM254v2/GCF_000002545.

# first 200 S. cerevisiae ORFs
zcat S_cerevisiae.ORFs.fasta.gz | awk '/^>/{n++} n<=200' > Scer_200.fasta

# C. glabrata CDS, renaming each sequence to its locus_tag, e.g. >CAGL0A00165g
zcat GCF_000002545.3_ASM254v2_cds_from_genomic.fna.gz | \
  sed -E 's/^>.*\[locus_tag=([^]]+)\].*/>\1/' > Cglabrata_cds.fasta

makeblastdb -in Cglabrata_cds.fasta -dbtype nucl -out Cglabrata_cds
blastn -task dc-megablast -query Scer_200.fasta -db Cglabrata_cds \
  -evalue 1e-5 -outfmt 6 -num_threads 4 -out Scer_vs_Cgla.blastn.tsv
wc -l Scer_vs_Cgla.blastn.tsv
head -3 Scer_vs_Cgla.blastn.tsv
```

```
     242 Scer_vs_Cgla.blastn.tsv
YAL001C CAGL0A00803g    83.607  61  10  0   2810    2870    2819    2879    3.16e-10    66.2
YAL003W CAGL0F08547g    82.137  627 103 3   1   621 1   624 3.42e-178   621
YAL004W CAGL0G03795g    85.781  647 92  0   2   648 671 25  0.0 753
```

(`dc-megablast` is a BLASTN mode designed for comparing DNA between species. For the amount of data here this takes under a second. This was run with BLAST+ 2.17.0.)

Reading a file with no header

If we just read the file, DuckDB can tell it is tab-delimited but has to make up column names:

```
.mode csv
SELECT * FROM read_csv('Scer_vs_Cgla.blastn.tsv', delim = '\t', header = false) LIMIT 2;
```

```
column00,column01,column02,column03,column04,column05,column06,column07,column08,column09,column10,column
YAL001C,CAGL0A00803g,83.607,61,10,0,2810,2870,2819,2879,3.16e-10,66.2
YAL003W,CAGL0F08547g,82.137,627,103,3,1,621,1,624,3.42e-178,621.0
```

So we use the `read_csv()` function and give it the 12 standard BLAST column names, and their types, with `columns = {...}`. We save it as a table called `blast` so we don't have to type this again:

```
.mode markdown
CREATE TABLE blast AS
SELECT * FROM read_csv('Scer_vs_Cgla.blastn.tsv',
    delim = '\t', header = false,
    columns = {
        'qseqid': 'VARCHAR', 'sseqid': 'VARCHAR', 'pident': 'DOUBLE',
        'length': 'INTEGER', 'mismatch': 'INTEGER', 'gapopen': 'INTEGER',
        'qstart': 'INTEGER', 'qend': 'INTEGER', 'sstart': 'INTEGER',
        'send': 'INTEGER', 'evalue': 'DOUBLE', 'bitscore': 'DOUBLE'});
```

The columns are: query ID, subject (database) ID, percent identity, alignment length, number of mismatches, number of gap openings, start and end of the alignment in the query, start and end in the subject, E-value, and bit score. `DOUBLE` is a decimal number.

How many queries have a hit?

Each line is one HSP (a local alignment). A query can hit several subjects, and hit the same subject more than once, so we count **distinct** IDs:

```
SELECT count(*)               AS n_hsps,
       count(DISTINCT qseqid) AS n_queries_with_hit,
       count(DISTINCT sseqid) AS n_subjects
FROM blast;
```

```
| n_hsps | n_queries_with_hit | n_subjects |
|-------:|-------------------:|-----------:|
| 242    | 119                | 156        |
```

119 of the 200 *S. cerevisiae* ORFs have a hit. Which queries have the most hits? (Genes in multi-gene families.)

```
SELECT qseqid, count(DISTINCT sseqid) AS n_subjects, count(*) AS n_hsps
FROM blast
GROUP BY qseqid
ORDER BY n_hsps DESC
LIMIT 5;
```

```
| qseqid  | n_subjects | n_hsps |
|---------|-----------:|-------:|
| YAL019W | 8          | 13     |
| YAL005C | 8          | 10     |
| YAL004W | 8          | 8      |
| YAR035W | 2          | 7      |
| YBL047C | 1          | 7      |
```

The best hit for each query

A very common task: keep only the top-scoring hit for each query. BLAST output is usually sorted by score within each query, but it is safer not to rely on that. DuckDB gives two easy ways.

`arg_max(x, y)` is an aggregate that returns the value of `x` from the row where `y` is largest. So "the `sseqid` with the highest `bitscore`, per query" is:

```
SELECT qseqid,
       arg_max(sseqid, bitscore) AS best_hit,
       max(bitscore)             AS best_bitscore
FROM blast
GROUP BY qseqid
ORDER BY qseqid
LIMIT 5;
```

```
| qseqid  |   best_hit   | best_bitscore |
|---------|--------------|--------------:|
| YAL001C | CAGL0A00803g | 66.2          |
| YAL003W | CAGL0F08547g | 621.0         |
| YAL004W | CAGL0G03795g | 753.0         |
| YAL005C | CAGL0G03795g | 2461.0        |
| YAL007C | CAGL0C02761g | 207.0         |
```

If you want the **whole row** of the best hit, use a **window function**. `row_number() OVER (PARTITION BY qseqid ORDER BY bitscore DESC)` numbers the hits 1, 2, 3… separately within each query (`PARTITION BY` is like `GROUP BY`, but keeps all the rows), from the highest bit score down. `QUALIFY` is like `WHERE` for window functions: keep only the rows numbered 1.

```
SELECT qseqid, sseqid, pident, length, evalue, bitscore
FROM blast
QUALIFY row_number() OVER (PARTITION BY qseqid ORDER BY bitscore DESC) = 1
ORDER BY qseqid
LIMIT 5;
```

```
| qseqid  |    sseqid    | pident | length |  evalue   | bitscore |
|---------|--------------|-------:|-------:|----------:|---------:|
| YAL001C | CAGL0A00803g | 83.607 | 61     | 3.16e-10  | 66.2     |
| YAL003W | CAGL0F08547g | 82.137 | 627    | 3.42e-178 | 621.0    |
| YAL004W | CAGL0G03795g | 85.781 | 647    | 0.0       | 753.0    |
| YAL005C | CAGL0G03795g | 88.336 | 1929   | 0.0       | 2461.0   |
| YAL007C | CAGL0C02761g | 69.014 | 568    | 2.58e-53  | 207.0    |
```

Notice that the best hit for `YAL001C` is only a 61 bp piece of a gene that is over 3,000 bp long - a reminder to look at alignment length, not just E-value.

Hits above 80% identity

```
SELECT qseqid, sseqid, pident, length
FROM blast
WHERE pident >= 80 AND length >= 300
ORDER BY pident DESC
LIMIT 5;
```

```
|  qseqid   |    sseqid    | pident | length |
|-----------|--------------|-------:|-------:|
| YAL038W   | CAGL0M12034g | 91.345 | 1502   |
| YAL037C-B | CAGL0M12034g | 91.173 | 963    |
| YBL027W   | CAGL0A03278g | 90.702 | 570    |
| YAL005C   | CAGL0G03795g | 88.336 | 1929   |
| YBL003C   | CAGL0C04411g | 87.719 | 399    |
```

```
SELECT count(DISTINCT qseqid) AS n FROM blast WHERE pident >= 80 AND length >= 300;
```

```
| n  |
|---:|
| 11 |
```

Finally, save the best hits as a new TSV file (with a header this time) for use in other programs:

```
COPY (SELECT qseqid, sseqid, pident, length, evalue, bitscore
      FROM blast
      QUALIFY row_number() OVER (PARTITION BY qseqid ORDER BY bitscore DESC) = 1
      ORDER BY qseqid)
  TO 'Scer_vs_Cgla.besthit.tsv' (HEADER, DELIMITER '\t');
```

```
head -3 Scer_vs_Cgla.besthit.tsv
```

```
qseqid  sseqid  pident  length  evalue  bitscore
YAL001C CAGL0A00803g    83.607  61  3.16e-10    66.2
YAL003W CAGL0F08547g    82.137  627 3.42e-178   621.0
```

The same queries work unchanged from Python with `duckdb.sql(...)`, and they work just as well on a BLAST table with millions of lines.

Exercises

1. Using `threatened-species.csv.gz`, find the 5 families with the most Critically Endangered (`CR`) species. Then find which of those families are plants and which are animals (hint: add `kingdom_name` to the `SELECT` and the `GROUP BY`).
2. How many species in the file have no common name (`main_common_name`)? What percentage of the total is that? Is it different for plants and animals?
3. In `airport-codes.csv.gz`, count the airports of each `type` in the US (`iso_country = 'US'`). Then use `iso_region` to find the 5 US states with the most heliports. Convert the file to Parquet and re-run your query with `.timer on` - how much faster is it?
4. Write a Python script that uses `duckdb` to read the BLAST table, finds the best hit for each query, and prints how many best hits have percent identity >= 80, between 60 and 80, and below 60 (hint: `CASE WHEN pident >= 80 THEN 'high' WHEN pident >= 60 THEN 'medium' ELSE 'low' END AS bin` and `GROUP BY bin`).
5. Re-run the BLAST search with all 6,713 *S. cerevisiae* ORFs as a SLURM job on the cluster. What fraction of yeast genes have a hit in *C. glabrata*? Make a small table of gene names from the FASTA headers (`grep '>' ... | awk ...` to get the ORF ID and gene name) and `JOIN` it to your best hits so the output shows gene names like `TFC3`.

Further reading

- DuckDB documentation - especially "Importing Data: CSV" and the SQL introduction
- DuckDB Python API
- SQL for Data Science (Data Carpentry) and Software Carpentry: Databases and SQL
- Apache Parquet
- Ziemann M, Eren Y, El-Osta A (2016) Gene name errors are widespread in the scientific literature. *Genome Biology* 17:177 doi:10.1186/s13059-016-1044-7