multiprocessio / **dsq**

Code | Issues 20 | Pull requests 2 | Actions | Projects | Security | Insights

Commandline tool for running SQL queries against JSON, CSV, Excel, Parquet, and more.

View license

3.8k stars **162** forks **29** watching Branches Activity Custom properties Tags

Public repository

main **5** Branches **29** Tags Go to file t Go to file + Add file Code

eatonphil Add read more c3ae0ba · 2 years ago

.github	Add golangci-lint step to Github workflows ...	3 years ago
scripts	Update datastation/runner mod and test ca...	3 years ago
testdata	Update datastation/runner mod and test ca...	3 years ago
.gitignore	Bump datastation (#85)	3 years ago
LICENSE.md	Update readme	4 years ago
README.md	Add read more	2 years ago
go.mod	Update datastation/runner mod and test ca...	3 years ago
go.sum	Update datastation/runner mod and test ca...	3 years ago
main.go	document -f/--file command line argument ...	2 years ago
sqlite.go	Add golangci-lint step to Github workflows ...	3 years ago

Not under active development

While development may continue in the future with a different architecture, for the moment you should probably instead use DuckDB, ClickHouse-local, or GlareDB (based on DataFusion).

These are built on stronger analytics foundations than projects like dsq based on SQLite. For example, column-oriented storage and vectorized execution, let alone JIT-compiled expression evaluation, are possible with these other projects.

More here.

Commandline tool for running SQL queries against JSON, CSV, Excel, Parquet, and more

Since Github doesn't provide a great way for you to learn about new releases and features, don't just star the repo, join the mailing list.

About

This is a CLI companion to DataStation (a GUI) for running SQL queries against data files. So if you want the GUI version of this, check out DataStation.

Install

Binaries for amd64 (x86_64) are provided for each release.

macOS Homebrew

`dsq` is available on macOS Homebrew:

```
$ brew install dsq
```

Binaries on macOS, Linux, WSL

On macOS, Linux, and WSL you can run the following:

```
$ VERSION="v0.23.0"
$ FILE="dsq-$(uname -s | awk '{ print tolower($0) }')-x64-$VERSION.zip"
$ curl -LO "https://github.com/multiprocessio/dsq/releases/download/$VERSION/$FILE"
$ unzip $FILE
$ sudo mv ./dsq /usr/local/bin/dsq
```

Or install manually from the releases page, unzip and add `dsq` to your `$PATH`.

Binaries on Windows (not WSL)

Download the latest Windows release, unzip it, and add `dsq` to your `$PATH`.

Build and install from source

If you are on another platform or architecture or want to grab the latest release, you can do so with Go 1.18+:

```
$ go install github.com/multiprocessio/dsq@latest
```

`dsq` will likely work on other platforms that Go is ported to such as AARCH64 and OpenBSD, but tests and builds are only run against x86_64 Windows/Linux/macOS.

Usage

You can either pipe data to `dsq` or you can pass a file name to it. NOTE: piping data doesn't work on Windows.

If you are passing a file, it must have the usual extension for its content type.

For example:

```
$ dsq testdata.json "SELECT * FROM {} WHERE x > 10"
```

Or:

```
$ dsq testdata.ndjson "SELECT name, AVG(time) FROM {} GROUP BY name ORDER BY AVG(time) DESC"
```

Pretty print

By default `dsq` prints ugly JSON. This is the most efficient mode.

```
$ dsq testdata/userdata.parquet 'select count(*) from {}'
[{"count(*)":1000}
]
```

If you want prettier JSON you can pipe `dsq` to `jq`.

```
$ dsq testdata/userdata.parquet 'select count(*) from {}' | jq
[
  {
    "count(*)": 1000
  }
]
```

Or you can enable pretty printing with `-p` or `--pretty` in `dsq` which will display your results in an ASCII table.

```
$ dsq --pretty testdata/userdata.parquet 'select count(*) from {}'
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
```

Piping data to dsq

When piping data to `dsq` you need to set the `-s` flag and specify the file extension or MIME type.

For example:

```
$ cat testdata.csv | dsq -s csv "SELECT * FROM {} LIMIT 1"
```

Or:

```
$ cat testdata.parquet | dsq -s parquet "SELECT COUNT(1) FROM {}"
```

Multiple files and joins

You can pass multiple files to DSQ. As long as they are supported data files in a valid format, you can run SQL against all files as tables. Each table can be accessed by the string `{N}` where `N` is the 0-based index of the file in the list of files passed on the commandline.

For example this joins two datasets of differing origin types (CSV and JSON).

```
$ dsq testdata/join/users.csv testdata/join/ages.json \
  "select {0}.name, {1}.age from {0} join {1} on {0}.id = {1}.id"
[{"age":88,"name":"Ted"},
{"age":56,"name":"Marjory"},
{"age":33,"name":"Micah"}]
```

You can also give file-table-names aliases since `dsq` uses standard SQL:

```
$ dsq testdata/join/users.csv testdata/join/ages.json \
  "select u.name, a.age from {0} u join {1} a on u.id = a.id"
[{"age":88,"name":"Ted"},
{"age":56,"name":"Marjory"},
{"age":33,"name":"Micah"}]
```

SQL query from file

As your query becomes more complex, it might be useful to store it in a file rather than specify it on the command line. To do so replace the query argument with `--file` or `-f` and the path to the file.

```
$ dsq data1.csv data2.csv -f query.sql
```

Transforming data to JSON without querying

As a shorthand for `dsq testdata.csv "SELECT * FROM {}"` to convert supported file types to JSON you can skip the query and the converted JSON will be dumped to stdout.

For example:

```
$ dsq testdata.csv
[{...some csv data...},{...some csv data...},...]
```

Array of objects nested within an object

DataStation and `dsq`'s SQL integration operates on an array of objects. If your array of objects happens to be at the top-level, you don't need to do anything. But if your array data is nested within an object you can add a "path" parameter to the table reference.

For example if you have this data:

```
$ cat api-results.json
{
  "data": {
    "data": [
      {"id": 1, "name": "Corah"},
      {"id": 3, "name": "Minh"}
    ]
  },
  "total": 2
}
```

You need to tell `dsq` that the path to the array data is `"data.data"`:

```
$ dsq --pretty api-results.json 'SELECT * FROM {0, "data.data"} ORDER BY id DESC'
+----+-------+
| id | name  |
+----+-------+
|  3 | Minh  |
|  1 | Corah |
+----+-------+
```

You can also use the shorthand `{"path"}` or `{'path'}` if you only have one table:

```
$ dsq --pretty api-results.json 'SELECT * FROM {"data.data"} ORDER BY id DESC'
+----+-------+
| id | name  |
+----+-------+
|  3 | Minh  |
|  1 | Corah |
+----+-------+
```

You can use either single or double quotes for the path.

Multiple Excel sheets

Excel files with multiple sheets are stored as an object with key being the sheet name and value being the sheet data as an array of objects.

If you have an Excel file with two sheets called `Sheet1` and `Sheet2` you can run `dsq` on the second sheet by specifying the sheet name as the path:

```
$ dsq data.xlsx 'SELECT COUNT(1) FROM {"Sheet2"}'
```

Limitation: nested arrays

You cannot specify a path through an array, only objects.

Nested object values

It's easiest to show an example. Let's say you have the following JSON file called `user_addresses.json`:

```
$ cat user_addresses.json
[
  {"name": "Agarrah", "location": {"city": "Toronto", "address": { "number": 1002 }}},
  {"name": "Minoara", "location": {"city": "Mexico City", "address": { "number": 19 }}},
  {"name": "Fontoon", "location": {"city": "New London", "address": { "number": 12 }}}
]
```

You can query the nested fields like so:

```
$ dsq user_addresses.json 'SELECT name, "location.city" FROM {}'
```

And if you need to disambiguate the table:

```
$ dsq user_addresses.json 'SELECT name, {}."location.city" FROM {}'
```

Caveat: PowerShell, CMD.exe

On PowerShell and CMD.exe you must escape inner double quotes with backslashes:

```
> dsq user_addresses.json 'select name, \"location.city\" from {}'
[{"location.city":"Toronto","name":"Agarrah"},
{"location.city":"Mexico City","name":"Minoara"},
{"location.city":"New London","name":"Fontoon"}]
```

Nested objects explained

Nested objects are collapsed and their new column name becomes the JSON path to the value connected by `.`. Actual dots in the path must be escaped with a backslash. Since `.` is a special character in SQL you must quote the whole new column name.

Limitation: whole object retrieval

You cannot query whole objects, you must ask for a specific path that results in a scalar value.

For example in the `user_addresses.json` example above you CANNOT do this:

```
$ dsq user_addresses.json 'SELECT name, {}."location" FROM {}'
```

Because `location` is not a scalar value. It is an object.

Nested arrays

Nested arrays are converted to a JSON string when stored in SQLite. Since SQLite supports querying JSON strings you can access that data as structured data even though it is a string.

So if you have data like this in `fields.json`:

```
[
  {"field1": [1]},
  {"field1": [2]},
]
```

You can request the entire field:

```
$ dsq fields.json "SELECT field1 FROM {}" | jq
[
  {
    "field1": "[1]"
  },
  {
    "field1": "[2]",
  }
]
```

JSON operators

You can get the first value in the array using SQL JSON operators.

```
$ dsq fields.json "SELECT field1->0 FROM {}" | jq
[
  {
    "field1->0": "1"
  },
  {
    "field1->0": "2"
  }
]
```

REGEXP

Since DataStation and `dsq` are built on SQLite, you can filter using `x REGEXP 'y'` where `x` is some column or value and `y` is a REGEXP string. SQLite doesn't pick a regexp implementation. DataStation and `dsq` use Go's regexp implementation which is more limited than PCRE2 because Go support for PCRE2 is not yet very mature.

```
$ dsq user_addresses.json "SELECT * FROM {} WHERE name REGEXP 'A.*'"
[{"location.address.number":1002,"location.city":"Toronto","name":"Agarrah"}]
```

Standard Library

dsq registers [go-sqlite3-stdlib](https://github.com/multiprocessio/go-sqlite3-stdlib) so you get access to numerous statistics, url, math, string, and regexp functions that aren't part of the SQLite base.

View that project docs for all available extended functions.

Output column order

When emitting JSON (i.e. without the `--pretty` flag) keys within an object are unordered.

If order is important to you you can filter with `jq`: `dsq x.csv 'SELECT a, b FROM {}' | jq --sort-keys`.

With the `--pretty` flag, column order is purely alphabetical. It is not possible at the moment for the order to depend on the SQL query order.

Dumping inferred schema

For any supported file you can dump the inferred schema rather than dumping the data or running a SQL query. Set the `--schema` flag to do this.

The inferred schema is very simple, only JSON types are supported. If the underlying format (like Parquet) supports finer-grained data types (like int64) this will not show up in the inferred schema. It will show up just as `number`.

For example:

```
$ dsq testdata/avro/test_data.avro --schema --pretty
Array of
  Object of
    birthdate of
      string
    cc of
      Varied of
        Object of
          long of
            number or
        Unknown
    comments of
      string
    country of
      string
    email of
      string
    first_name of
      string
    gender of
      string
    id of
      number
    ip_address of
      string
    last_name of
      string
    registration_dttm of
      string
    salary of
      Varied of
        Object of
          double of
            number or
        Unknown
    title of
      string
```

You can print this as a structured JSON string by omitting the `--pretty` flag when setting the `--schema` flag.

Caching

Sometimes you want to do some exploration on a dataset that isn't changing frequently. By turning on the `--cache` or `-C` flag DataStation will store the imported data on disk and not delete it when the run is over.

With caching on, DataStation calculates a SHA1 sum of all the files you specified. If the sum ever changes then it will reimport all the files. Otherwise when you run additional queries with the cache flag on it will reuse that existing database and not reimport the files.

Since without caching on DataStation uses an in-memory database, the initial query with caching on may take slightly longer than with caching off. Subsequent queries will be substantially faster though (for large datasets).

For example, in the first run with caching on this query might take 30s:

```
$ dsq some-large-file.json --cache 'SELECT COUNT(1) FROM {}'
```

But when you run another query it might only take 1s.

```
$ dsq some-large-file.json --cache 'SELECT SUM(age) FROM {}'
```

Not because we cache any result but because we cache importing the file into SQLite.

So even if you change the query, as long as the file doesn't change, the cache is effective.

To make this permanent you can export `DSQ_CACHE=true` in your environment.

Interactive REPL

Use the `-i` or `--interactive` flag to enter an interactive REPL where you can run multiple SQL queries.

```
$ dsq some-large-file.json -i
dsq> SELECT COUNT(1) FROM {};
+----------+
| COUNT(1) |
+----------+
|     1000 |
+----------+
(1 row)
dsq> SELECT * FROM {} WHERE NAME = 'Kevin';
(0 rows)
```

Converting numbers in CSV and TSV files

CSV and TSV files do not allow to specify the type of the individual values contained in them. All values are treated as strings by default.

This can lead to unexpected results in queries. Consider the following example:

```
$ cat scores.csv
name,score
Fritz,90
Rainer,95.2
Fountainer,100

$ dsq scores.csv "SELECT * FROM {} ORDER BY score"
[{"name":"Fountainer","score":"100"},
{"name":"Fritz","score":"90"},
{"name":"Rainer","score":"95.2"}]
```

Note how the `score` column contains numerical values only. Still, sorting by that column yields unexpected results because the values are treated as strings, and sorted lexically. (You can tell that the individual scores were imported as strings because they're quoted in the JSON result.)

Use the `-n` or `--convert-numbers` flag to auto-detect and convert numerical values (integers and floats) in imported files:

```
$ dsq ~/scores.csv --convert-numbers "SELECT * FROM {} ORDER BY score"
[{"name":"Fritz","score":90},
{"name":"Rainer","score":95.2},
{"name":"Fountainer","score":100}]
```

Note how the scores are imported as numbers now and how the records in the result set are sorted by their numerical value. Also note that the individual scores are no longer quoted in the JSON result.

To make this permanent you can export `DSQ_CONVERT_NUMBERS=true` in your environment. Turning this on disables some optimizations.

Supported Data Types

Name	File Extension(s)	Mime Type	Notes
CSV	`csv`	`text/csv`	
TSV	`tsv`, `tab`	`text/tab-separated-values`	
JSON	`json`	`application/json`	Must be an array of objects or a [path to an array of objects]().
Newline-delimited JSON	`ndjson`, `jsonl`	`application/jsonlines`	
Concatenated JSON	`cjson`	`application/jsonconcat`	
ORC	`orc`	`orc`	
Parquet	`parquet`	`parquet`	
Avro	`avro`	`application/avro`	
YAML	`yaml`, `yml`	`application/yaml`	
Excel	`xlsx`, `xls`	`application/vnd.ms-excel`	If you have multiple sheets, you must [specify a sheet path]().

Name	File Extension(s)	Mime Type	Notes
ODS	`ods`	`application/vnd.oasis.opendocument.spreadsheet`	If you have multiple sheets, you must specify a sheet path.
Apache Error Logs	NA	`text/apache2error`	Currently only works if being piped in.
Apache Access Logs	NA	`text/apache2access`	Currently only works if being piped in.
Nginx Access Logs	NA	`text/nginxaccess`	Currently only works if being piped in.
LogFmt Logs	`logfmt`	`text/logfmt`	

Engine

Under the hood dsq uses DataStation as a library and under that hood DataStation uses SQLite to power these kinds of SQL queries on arbitrary (structured) data.

Comparisons

Name	Link	Caching	Engine	Supported File Types	Binary Size
dsq	Here	Yes	SQLite	CSV, TSV, a few variations of JSON, Parquet, Excel, ODS (OpenOffice Calc), ORC, Avro, YAML, Logs	49M
q	http://harelba.github.io/q/	Yes	SQLite	CSV, TSV	82M
textql	https://github.com/dinedal/textql	No	SQLite	CSV, TSV	7.3M
octoql	https://github.com/cube2222/octosql	No	Custom engine	JSON, CSV, Excel, Parquet	18M
csvq	https://github.com/mithrandie/csvq	No	Custom engine	CSV	15M
utils	utils				binary
trdsql	https://github.com/noborus/trdsql	No	SQLite, MySQL or PostgreSQL	Few variations of JSON, TSV, LTSV, TBLN, CSV	14M
spysql	https://github.com/dcmoura/spyql	No	Custom engine	CSV, JSON, TEXT	N/A, Not a single binary
duckdb	https://github.com/duckdb/duckdb	?	Custom engine	CSV, Parquet	35M

Not included:

- clickhouse-local: fastest of anything listed here but so gigantic (over 2GB) that it can't reasonably be considered a good tool for any environment
- sqlite3: requires multiple commands to ingest CSV, not great for one-liners
- datafusion-cli: very fast (slower only than clickhouse-local) but requires multiple commands to ingest CSV, so not great for one-liners

Benchmark

This benchmark was run June 19, 2022. It is run on a dedicated bare metal instance on OVH with:

- 64 GB DDR4 ECC 2,133 MHz
- 2x450 GB SSD NVMe in Soft RAID
- Intel Xeon E3-1230v6 - 4c/8t - 3.5 GHz/3.9 GHz

It runs a `SELECT passenger_count, COUNT(*), AVG(total_amount) FROM taxi.csv GROUP BY passenger_count` query against the well-known NYC Yellow Taxi Trip Dataset. Specifically, the CSV file from April 2021 is used. It's a 200MB CSV file with ~2 million rows, 18 columns, and mostly numerical values.

The script is here. It is an adaptation of the benchmark that the octosql devs run.

Program	Version	Mean [s]	Min [s]	Max [s]	Relative
dsq	0.20.1 (caching on)	1.151 ± 0.010	1.131	1.159	1.00
duckdb	0.3.4	1.723 ± 0.023	1.708	1.757	1.50 ± 0.02
octosql	0.7.3	2.005 ± 0.008	1.991	2.015	1.74 ± 0.02
q	3.1.6 (caching on)	2.028 ± 0.010	2.021	2.055	1.76 ± 0.02
sqlite3 *	3.36.0	4.204 ± 0.018	4.177	4.229	3.64 ± 0.04
trdsql	0.10.0	12.972 ± 0.225	12.554	13.392	11.27 ± 0.22
dsq	0.20.1 (default)	15.030 ± 0.086	14.895	15.149	13.06 ± 0.13
textql	fca00ec	19.148 ± 0.183	18.865	19.500	16.63 ± 0.21
spyql	0.6.0	16.985 ± 0.105	16.854	17.161	14.75 ± 0.16
q	3.1.6 (default)	24.061 ± 0.095	23.954	24.220	20.90 ± 0.20

* While dsq and q are built on top of sqlite3 there is not a builtin way in sqlite3 to cache ingested files without a bit of scripting

Not included:

- clickhouse-local: faster than any of these but over 2GB so not a reasonable general-purpose CLI
- datafusion-cli: slower only than clickhouse-local but requires multiple commands to ingest CSV, can't do one-liners
- sqlite-utils: takes minutes to finish

Notes

OctoSQL, duckdb, and SpyQL implement their own SQL engines. dsq, q, trdsql, and textql copy data into SQLite and depend on the SQLite engine for query execution.

Tools that implement their own SQL engines can do better on 1) ingestion and 2) queries that act on a subset of data (such as limited columns or limited rows). These tools implement ad-hoc subsets of SQL that may be missing or differ from your favorite syntax. On the other hand, tools that depend on SQLite have the benefit of providing a well-tested and well-documented SQL engine. DuckDB is exceptional since there is a dedicated company behind it.

dsq also comes with numerous useful functions (e.g. best-effort date parsing, URL parsing/extraction, statistics functions, etc.) on top of SQLite builtins.

Third-party integrations

- ob-dsq

Community

Join us at #dsq on the Multiprocess Discord.

Releases 29

v0.23.0 Latest
on Oct 21, 2022

+ 28 releases

Packages

No packages published

Contributors 15

Languages

- **Go** 49.3%
- **Python** 35.9%
- **JavaScript** 7.4%
- **Shell** 5.7%
- **PowerShell** 1.7%