

pg_parquet

Copy from/to Parquet files in PostgreSQL!

`pg_parquet` is a PostgreSQL extension that allows you to read and write Parquet files, which are located in S3 or file system, from PostgreSQL via COPY TO/FROM commands. It depends on Apache Arrow project to read and write Parquet files and pgrx project to extend PostgreSQL's COPY command.

```
-- Copy a query result into Parquet in S3
COPY (SELECT * FROM table) TO 's3://mybucket/data.parquet' WITH (format 'parquet');

-- Load data from Parquet in S3
COPY table FROM 's3://mybucket/data.parquet' WITH (format 'parquet');
```

Quick Reference

- Installation From Source
- Usage
 - Copy FROM/TO Parquet files TO/FROM Postgres tables
 - Inspect Parquet schema
 - Inspect Parquet metadata
- Object Store Support
- Copy Options
- Configuration
- Supported Types
 - Nested Types
- Postgres Support Matrix

Installation From Source

After installing Postgres, you need to set up rustup, cargo-pgrx to build the extension.

```
# install rustup
> curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

# install cargo-pgrx
> cargo install cargo-pgrx

# configure pgrx
> cargo pgrx init --pg17 $(which pg_config)

# append the extension to shared_preload_libraries in ~/.pgrx/data-17/postgresql.conf
> echo "shared_preload_libraries = 'pg_parquet'" >> ~/.pgrx/data-17/postgresql.conf

# run cargo-pgrx to build and install the extension
> cargo pgrx run
```

```
# create the extension in the database
psql> "CREATE EXTENSION pg_parquet;"
```

Usage

There are mainly 3 things that you can do with `pg_parquet`: 1. You can export Postgres tables/queries to Parquet files, 2. You can ingest data from Parquet files to Postgres tables, 3. You can inspect the schema and metadata of Parquet files.

COPY to/from Parquet files from/to Postgres tables

You can use PostgreSQL's `COPY` command to read and write Parquet files. Below is an example of how to write a PostgreSQL table, with complex types, into a Parquet file and then to read the Parquet file content back into the same table.

```
-- create composite types
CREATE TYPE product_item AS (id INT, name TEXT, price float4);
CREATE TYPE product AS (id INT, name TEXT, items product_item[]);

-- create a table with complex types
CREATE TABLE product_example (
    id int,
    product product,
    products product[],
    created_at TIMESTAMP,
    updated_at TIMESTAMPTZ
);

-- insert some rows into the table
insert into product_example values (
    1,
    ROW(1, 'product 1', ARRAY[ROW(1, 'item 1', 1.0), ROW(2, 'item 2', 2.0), NULL]::product_item[]),
    ARRAY[ROW(1, NULL, NULL)::product, NULL],
    now(),
    '2022-05-01 12:00:00-04'
);

-- copy the table to a parquet file
COPY product_example TO '/tmp/product_example.parquet' (format 'parquet', compression 'gzip');

-- show table
SELECT * FROM product_example;

-- copy the parquet file to the table
COPY product_example FROM '/tmp/product_example.parquet';
```

```
-- show table
SELECT * FROM product_example;
```

Inspect Parquet schema

You can call `SELECT * FROM parquet.schema(<uri>)` to discover the schema of the Parquet file at given uri.

```
SELECT * FROM parquet.schema('/tmp/product_example.parquet') LIMIT 10;
```

| uri | name | type_name | type_length | repetition_type |
|------------------------------|--------------|------------|-------------|-----------------|
| /tmp/product_example.parquet | arrow_schema | | | |
| /tmp/product_example.parquet | id | INT32 | | OPTIONAL |
| /tmp/product_example.parquet | product | | | OPTIONAL |
| /tmp/product_example.parquet | id | INT32 | | OPTIONAL |
| /tmp/product_example.parquet | name | BYTE_ARRAY | | OPTIONAL |
| /tmp/product_example.parquet | items | | | OPTIONAL |
| /tmp/product_example.parquet | list | | | REPEATED |
| /tmp/product_example.parquet | element | | | OPTIONAL |
| /tmp/product_example.parquet | id | INT32 | | OPTIONAL |
| /tmp/product_example.parquet | name | BYTE_ARRAY | | OPTIONAL |

(10 rows)

Inspect Parquet metadata

You can call `SELECT * FROM parquet.metadata(<uri>)` to discover the detailed metadata of the Parquet file, such as column statistics, at given uri.

```
SELECT uri, row_group_id, row_group_num_rows, row_group_num_columns, row_group_bytes, column...
```

| uri | row_group_id | row_group_num_rows | row_group_num_columns |
|------------------------------|--------------|--------------------|-----------------------|
| /tmp/product_example.parquet | 0 | 1 | 13 |

(1 row)

```
SELECT stats_null_count, stats_distinct_count, stats_min, stats_max, compression, encodings...
```

| stats_null_count | stats_distinct_count | stats_min | stats_max | compression |
|------------------|----------------------|-----------|-----------|--------------------|
| 0 | 1 | 1 | 1 | GZIP(GzipLevel(6)) |

(1 row)

You can call `SELECT * FROM parquet.file_metadata(<uri>)` to discover file level metadata of the Parquet file, such as format version, at given uri.

```
SELECT * FROM parquet.file_metadata('/tmp/product_example.parquet')
```

| uri | created_by | num_rows | num_row_groups | format_version |
|------------------------------|------------|----------|----------------|----------------|
| /tmp/product_example.parquet | pg_parquet | 1 | 1 | 1 |

(1 row)

You can call `SELECT * FROM parquet.kv_metadata(<uri>)` to query custom key-value metadata of the Parquet file at given uri.

```
SELECT uri, encode(key, 'escape') as key, encode(value, 'escape') as value FROM parquet.kv_m
      uri                |      key                |      value
-----+-----+-----
 /tmp/product_example.parquet | ARROW:schema | //5gIAAAQAAAA ...
(1 row)
```

Object Store Support

`pg_parquet` supports reading and writing Parquet files from/to S3 object store. Only the uris with `s3://` scheme is supported.

The simplest way to configure object storage is by creating the standard `~/.aws/credentials` and `~/.aws/config` files:

```
$ cat ~/.aws/credentials
[default]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

```
$ cat ~/.aws/config
[default]
region = eu-central-1
```

Alternatively, you can use the following environment variables when starting postgres to configure the S3 client: - `AWS_ACCESS_KEY_ID`: the access key ID of the AWS account - `AWS_SECRET_ACCESS_KEY`: the secret access key of the AWS account - `AWS_SESSION_TOKEN`: the session token for the AWS account - `AWS_REGION`: the default region of the AWS account - `AWS_ENDPOINT_URL`: the endpoint - `AWS_SHARED_CREDENTIALS_FILE`: an alternative location for the credentials file (**only via environment variables**) - `AWS_CONFIG_FILE`: an alternative location for the config file (**only via environment variables**) - `AWS_PROFILE`: the name of the profile from the credentials and config file (default profile name is `default`) (**only via environment variables**) - `AWS_ALLOW_HTTP`: allows http endpoints (**only via environment variables**)

[!NOTE] To be able to write into a object store location, you need to grant `parquet_object_store_write` role to your current postgres user. Similarly, to read from an object store location, you need to grant `parquet_object_store_read` role to your current postgres user.

Copy Options

`pg_parquet` supports the following options in the `COPY TO` command: - `format parquet`: you need to specify this option to read or write Parquet files which

does not end with `.parquet[.<compression>]` extension, `- row_group_size <int>`: the number of rows in each row group while writing Parquet files. The default row group size is 122880, `- row_group_size_bytes <int>`: the total byte size of rows in each row group while writing Parquet files. The default row group size bytes is `row_group_size * 1024`, `- compression <string>`: the compression format to use while writing Parquet files. The supported compression formats are `uncompressed`, `snappy`, `gzip`, `brotli`, `lz4`, `lz4raw` and `zstd`. The default compression format is `snappy`. If not specified, the compression format is determined by the file extension, `- compression_level <int>`: the compression level to use while writing Parquet files. The supported compression levels are only supported for `gzip`, `zstd` and `brotli` compression formats. The default compression level is 6 for `gzip` (0-10), 1 for `zstd` (1-22) and 1 for `brotli` (0-11).

`pg_parquet` supports the following options in the `COPY FROM` command: `- format parquet`: you need to specify this option to read or write Parquet files which does not end with `.parquet[.<compression>]` extension, `- match_by <string>`: method to match Parquet file fields to PostgreSQL table columns. The available methods are `position` and `name`. The default method is `position`. You can set it to `name` to match the columns by their name rather than by their position in the schema (default). Match by `name` is useful when field order differs between the Parquet file and the table, but their names match.

Configuration

There is currently only one GUC parameter to enable/disable the `pg_parquet`: `- pg_parquet.enable_copy_hooks`: you can set this parameter to `on` or `off` to enable or disable the `pg_parquet` extension. The default value is `on`.

Supported Types

`pg_parquet` has rich type support, including PostgreSQL's primitive, array, and composite types. Below is the table of the supported types in PostgreSQL and their corresponding Parquet types.

| PostgreSQL Type | Parquet Physical Type | Logical Type |
|-------------------------|---------------------------------------|---------------------------|
| <code>bool</code> | <code>BOOLEAN</code> | |
| <code>smallint</code> | <code>INT16</code> | |
| <code>integer</code> | <code>INT32</code> | |
| <code>bigint</code> | <code>INT64</code> | |
| <code>real</code> | <code>FLOAT</code> | |
| <code>oid</code> | <code>INT32</code> | |
| <code>double</code> | <code>DOUBLE</code> | |
| <code>numeric(1)</code> | <code>FIXED_LEN_BYTE_ARRAY(16)</code> | <code>DECIMAL(128)</code> |
| <code>text</code> | <code>BYTE_ARRAY</code> | <code>STRING</code> |

| PostgreSQL Type | Parquet Physical Type | Logical Type |
|-----------------------------|-------------------------|-------------------------------|
| <code>json</code> | <code>BYTE_ARRAY</code> | <code>STRING</code> |
| <code>bytea</code> | <code>BYTE_ARRAY</code> | |
| <code>date (2)</code> | <code>INT32</code> | <code>DATE</code> |
| <code>timestamp</code> | <code>INT64</code> | <code>TIMESTAMP_MICROS</code> |
| <code>timestampz (3)</code> | <code>INT64</code> | <code>TIMESTAMP_MICROS</code> |
| <code>time</code> | <code>INT64</code> | <code>TIME_MICROS</code> |
| <code>timetz(3)</code> | <code>INT64</code> | <code>TIME_MICROS</code> |
| <code>geometry(4)</code> | <code>BYTE_ARRAY</code> | |

Nested Types

| PostgreSQL Type | Parquet Physical Type | Logical Type |
|-----------------------------|-------------------------|---------------------|
| <code>composite</code> | <code>GROUP</code> | <code>STRUCT</code> |
| <code>array</code> | element's physical type | <code>LIST</code> |
| <code>crunchy_map(5)</code> | <code>GROUP</code> | <code>MAP</code> |

[!WARNING] - (1) `numeric` type is written the smallest possible memory width to parquet file as follows: * `numeric(P <= 9, S)` is represented as `INT32` with `DECIMAL` logical type * `numeric(9 < P <= 18, S)` is represented as `INT64` with `DECIMAL` logical type * `numeric(18 < P <= 38, S)` is represented as `FIXED_LEN_BYTE_ARRAY(9-16)` with `DECIMAL` logical type * `numeric(38 < P, S)` is represented as `BYTE_ARRAY` with `STRING` logical type * `numeric` is allowed by Postgres. (precision and scale not specified). These are represented by a default precision (38) and scale (9) instead of writing them as string. You get runtime error if your table tries to read or write a numeric value which is not allowed by the default precision and scale (29 integral digits before decimal point, 9 digits after decimal point). - (2) The `date` type is represented according to `Unix epoch` when writing to Parquet files. It is converted back according to `PostgreSQL epoch` when reading from Parquet files. - (3) The `timestampz` and `timetz` types are adjusted to `UTC` when writing to Parquet files. They are converted back with `UTC` timezone when reading from Parquet files. - (4) The `geometry` type is represented as `BYTE_ARRAY` encoded as `WKB` when `postgis` extension is created. Otherwise, it is represented as `BYTE_ARRAY` with `STRING` logical type. - (5) `crunchy_map` is dependent on functionality provided by Crunchy Bridge. The `crunchy_map` type is represented as `GROUP` with `MAP` logical type when `crunchy_map` extension is created. Otherwise, it is represented as `BYTE_ARRAY` with `STRING` logical type.

[!WARNING] Any type that does not have a corresponding Parquet type will be represented, as a fallback mechanism, as `BYTE_ARRAY` with `STRING` logical type. e.g. `enum`

Postgres Support Matrix

`pg_parquet` supports the following PostgreSQL versions: | PostgreSQL Major
Version | Supported | |-----|-----| | 14 | | 15 | | 16 | |
17 | |