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`, `Azure Blob Storage`, `Google Cloud Storage`, `http(s) endpoints` 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.

```sql
-- 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
  - Inspect Parquet column statistics
- 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.

```bash
# 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
```

```bash
# initialize a data directory, build and install the extension (to the targets specified by
> cargo pgrx run
# alternatively you can only build and install the extension (pass --release flag for produ
> cargo pgrx install --release

# 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 from/to 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.

```sql
-- 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_i
    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'
```

```sql
-- 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;
```

You can also use `COPY` command to read and write Parquet stream from/to standard input and output. Below is an example usage (you have to specify `format = parquet`):

```bash
psql -d pg_parquet -p 28817 -h localhost -c "create table product_example_reconstructed (lik
 CREATE TABLE

psql -d pg_parquet -p 28817 -h localhost -c "copy product_example to stdout (format parquet)
 COPY 2
```

Inspect Parquet schema

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

```sql
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.

```sql
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         | GZIP(GzipLevel(6)) | PLAI
(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)
```

Inspect Parquet column statistics

You can call `SELECT * FROM parquet.column_stats(<uri>)` to discover the column statistics of the Parquet file, such as min and max value for the column, at given uri.

```
SELECT * FROM parquet.column_stats('/tmp/product_example.parquet')
 column_id | field_id |         stats_min          |         stats_max          | stats_nul
-----------+----------+----------------------------+----------------------------+----------
         4 |        7 | item 1                     | item 2                     |
         6 |       11 | 1                          | 1                          |
         7 |       12 |                            |                            |
        10 |       17 |                            |                            |
         0 |        0 | 1                          | 1                          |
        11 |       18 | 2025-03-11 14:01:22.045739 | 2025-03-11 14:01:22.045739 |
         3 |        6 | 1                          | 2                          |
        12 |       19 | 2022-05-01 19:00:00+03     | 2022-05-01 19:00:00+03     |
         8 |       15 |                            |                            |
         5 |        8 | 1                          | 2                          |
         9 |       16 |                            |                            |
         1 |        2 | 1                          | 1                          |
```

```
         2 |           3 | product 1                         | product 1                         |
(13 rows)
```

Object Store Support

`pg_parquet` supports reading and writing Parquet files from/to `S3`, `Azure Blob Storage`, `http(s)` and `Google Cloud Storage` object stores.

> [!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.

S3 Storage 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)**

Config source priority order is shown below: 1. Environment variables, 2. Config file.

Supported S3 uri formats are shown below: - s3:// <bucket> / <path> - https:// <bucket>.s3.amazonaws.com / <path> - https:// s3.amazonaws.com / <bucket> / <path>

Supported authorization methods' priority order is shown below: 1. Temporary session tokens by assuming roles, 2. Long term credentials.

Azure Blob Storage The simplest way to configure object storage is by creating the standard `~/.azure/config` file:

```
$ cat ~/.azure/config
[storage]
account = devstoreaccount1
key = Eby8vdM02xNOcqFlqUwJPLlmEtlCDXJ1OUzFT50uSRZ6IFsuFq2UVErCz4I6tq/K1SZFPTOtr/KBHBeksoGMGw
```

Alternatively, you can use the following environment variables when starting postgres to configure the Azure Blob Storage client: - `AZURE_STORAGE_ACCOUNT`: the storage account name of the Azure Blob - `AZURE_STORAGE_KEY`: the storage key of the Azure Blob - `AZURE_STORAGE_CONNECTION_STRING`: the connection string for the Azure Blob (overrides any other config) - `AZURE_STORAGE_SAS_TOKEN`: the storage SAS token for the Azure Blob - `AZURE_TENANT_ID`: the tenant id for client secret auth **(only via environment variables)** - `AZURE_CLIENT_ID`: the client id for client secret auth **(only via environment variables)** - `AZURE_CLIENT_SECRET`: the client secret for client secret auth **(only via environment variables)** - `AZURE_STORAGE_ENDPOINT`: the endpoint **(only via environment variables)** - `AZURE_CONFIG_FILE`: an alternative location for the config file **(only via environment variables)** - `AZURE_ALLOW_HTTP`: allows http endpoints **(only via environment variables)**

Config source priority order is shown below: 1. Connection string (read from environment variable or config file), 2. Environment variables, 3. Config file.

Supported Azure Blob Storage uri formats are shown below: - az:// <container> / <path> - azure:// <container> / <path> - https:// <account>.blob.core.windows.net / <container>

Supported authorization methods' priority order is shown below: 1. Bearer token via client secret, 2. Sas token, 3. Storage key.

Http(s) Storage `Https` uris are supported by default. You can set `ALLOW_HTTP` environment variable to allow `http` uris.

Google Cloud Storage The simplest way to configure object storage is by creating a json config file like [`~/.config/gcloud/application_default_credentials.json`] (can be generated by `gcloud auth application-default login`):

```
$ cat ~/.config/gcloud/application_default_credentials.json
{
  "gcs_base_url": "http://localhost:4443",
  "disable_oauth": true,
  "client_email": "",
  "private_key_id": "",
  "private_key": ""
}
```

Alternatively, you can use the following environment variables when starting postgres to configure the Google Cloud Storage client: - `GOOGLE_SERVICE_ACCOUNT_KEY`: json serialized service account key **(only via environment variables)** - `GOOGLE_SERVICE_ACCOUNT_PATH`: an alternative location for the config file **(only via environment variables)**

Supported Google Cloud Storage uri formats are shown below: - gs:// <bucket> / <path>

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, - `file_size_bytes <string>`: the total file size per Parquet file. When set, the parquet files, with target size, are created under parent directory (named the same as file name). By default, when not specified, a single file is generated without creating a parent folder. You can specify total bytes without unit like `file_size_bytes 2000000` or with unit (KB, MB, or GB) like `file_size_bytes '1MB'`, - `field_ids <string>`: fields ids that are assigned to the fields in Parquet file schema. By default, no field ids are assigned. Pass `auto` to let pg_parquet generate field ids. You can pass a json string to explicitly pass the field ids, - `row_group_size <int64>`: the number of rows in each row group while writing Parquet files. The default row group size is `122880`, - `row_group_size_bytes <int64>`: 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
`bool`	BOOLEAN	
`smallint`	INT16	
`integer`	INT32	
`bigint`	INT64	
`real`	FLOAT	
`oid`	INT32	
`double`	DOUBLE	
`numeric`(1)	FIXED_LEN_BYTE_ARRAY(16)	DECIMAL(128)
`text`	BYTE_ARRAY	STRING
`json`	BYTE_ARRAY	JSON
`jsonb`	BYTE_ARRAY	JSON
`uuid`	FIXED_LEN_BYTE_ARRAY(16)	UUID
`bytea`	BYTE_ARRAY	
`date` (2)	INT32	DATE
`timestamp`	INT64	TIMESTAMP_MICROS
`timestamptz` (3)	INT64	TIMESTAMP_MICROS
`time`	INT64	TIME_MICROS
`timetz`(3)	INT64	TIME_MICROS
`geometry`(4)	BYTE_ARRAY	

Nested Types

PostgreSQL Type	Parquet Physical Type	Logical Type
`composite`	GROUP	STRUCT
`array`	element's physical type	LIST
`crunchy_map`(5)	GROUP	MAP

> [!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 `timestamptz` 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`, specified by geoparquet spec, 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 | |