Package 'nanoparquet'

September 16, 2026

Title Read and Write 'Parquet' Files

Version 0.5.2

Description Self-sufficient reader and writer for flat 'Parquet' files. Can read most 'Parquet' data types. Can write many 'R' data types, including factors and temporal types. See docs for limitations.

Depends R (>= 4.0.0)

License MIT + file LICENSE

URL https://github.com/r-lib/nanoparquet, https://nanoparquet.r-lib.org/

BugReports https://github.com/r-lib/nanoparquet/issues

Encoding UTF-8

Suggests arrow, bit64, blob, DBI, duckdb (>= 1.4.0), hms, mockery, pillar, processx, rprojroot, spelling, testthat, tzdb, withr

Config/testthat/edition 3

Config/testthat/parallel TRUE

Config/Needs/website tidyverse/tidytemplate, r-lib/pkgdown, dplyr, gt, gtExtras, knitr, nycflights13, prettyunits, quarto, rmarkdown, sessioninfo, svglite

Language en-US

Biarch true

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Gábor Csárdi [aut, cre],
Hannes Mühleisen [aut, cph] (ORCID:
<https://orcid.org/0000-0001-8552-0029>),
Google Inc. [cph],
Apache Software Foundation [cph],
Posit Software, PBC [cph],
RAD Game Tools [cph],
Valve Software [cph],

Tenacious Software LLC [cph],
Facebook, Inc. [cph]

Maintainer Gábor Csárdi <csardi.gabor@gmail.com>

Repository CRAN

Date/Publication 2026-09-16 03:30:02 UTC

Contents

nanoparquet-package 2
append_parquet 5
edit_parquet_metadata 6
infer_parquet_schema 7
nanoparquet-types 8
parquet-encodings 13
parquet_column_types 15
parquet_options 16
parquet_schema 18
read_parquet 20
read_parquet_info 21
read_parquet_metadata 22
read_parquet_schema 24
write_parquet 25

Index **28**

`nanoparquet-package` *nanoparquet: Read and Write 'Parquet' Files*

Description

Self-sufficient reader and writer for flat 'Parquet' files. Can read most 'Parquet' data types. Can write many 'R' data types, including factors and temporal types. See docs for limitations.

Details

`nanoparquet` is a reader and writer for a common subset of Parquet files.

Features::

- Read and write flat (i.e. non-nested) Parquet files.
- Can read most Parquet data types.
- Can read a subset of columns from a Parquet file.
- Can write many R data types, including factors and temporal types to Parquet.
- Can append a data frame to a Parquet file without first reading and then rewriting the whole file.
- Completely dependency free.

- Supports Snappy, Gzip and Zstd compression.
- Competitive with other tools in terms of speed, memory use and file size.

Limitations::

- Nested Parquet types are not supported.
- Some Parquet logical types are not supported: `INTERVAL`, `UNKNOWN`.
- Only Snappy, Gzip and Zstd compression is supported.
- Encryption is not supported.
- Reading files from URLs is not supported.
- nanoparquet always reads the data (or the selected subset of it) into memory. It does not work with out-of-memory data in Parquet files like Apache Arrow and DuckDB does.

Installation:

Install the R package from CRAN:

```
install.packages("nanoparquet")
```

Usage:

Read:

Call `read_parquet()` to read a Parquet file:

```
df <- nanoparquet::read_parquet("example.parquet")
```

To see the columns of a Parquet file and how their types are mapped to R types by `read_parquet()`, call `read_parquet_schema()` first:

```
nanoparquet::read_parquet_schema("example.parquet")
```

Folders of similar-structured Parquet files (e.g. produced by Spark) can be read like this:

```
df <- data.table::rbindlist(lapply(
  Sys.glob("some-folder/part-*.parquet"),
  nanoparquet::read_parquet
))
```

Write:

Call `write_parquet()` to write a data frame to a Parquet file:

```
nanoparquet::write_parquet(mtcars, "mtcars.parquet")
```

To see how the columns of the data frame will be mapped to Parquet types by `write_parquet()`, call `infer_parquet_schema()` first:

```
nanoparquet::infer_parquet_schema(mtcars)
```

Inspect:

Call `read_parquet_info()`, `read_parquet_schema()`, or `read_parquet_metadata()` to see various kinds of metadata from a Parquet file:

- `read_parquet_info()` shows a basic summary of the file.
- `read_parquet_schema()` shows all columns, including non-leaf columns, and how they are mapped to R types by `read_parquet()`.
- `read_parquet_metadata()` shows the most complete metadata information: file meta data, the schema, the row groups and column chunks of the file.

```
nanoparquet::read_parquet_info("mtcars.parquet")
nanoparquet::read_parquet_schema("mtcars.parquet")
nanoparquet::read_parquet_metadata("mtcars.parquet")
```

If you find a file that should be supported but isn't, please open an issue here with a link to the file.

Options:

See also `?parquet_options()` for further details.

- `nanoparquet.class`: extra class to add to data frames returned by `read_parquet()`. If it is not defined, the default is `"tbl"`, which changes how the data frame is printed if the pillar package is loaded.
- `nanoparquet.compression_level`: See `?parquet_options()` for the defaults and the possible values for each compression method. `Inf` selects maximum compression for each method.
- `nanoparquet.num_rows_per_row_group`: The number of rows to put into a row group by `write_parquet()`, if row groups are not specified explicitly. It should be an integer scalar. Defaults to 122,880, the same as DuckDB's default.
- `nanoparquet.use_arrow_metadata`: unless this is set to `FALSE`, `read_parquet()` will make use of Arrow metadata in the Parquet file. Currently this is used to detect factor columns.
- `nanoparquet.write_arrow_metadata`: unless this is set to `FALSE`, `write_parquet()` will add Arrow metadata to the Parquet file. This helps preserving classes of columns, e.g. factors will be read back as factors, both by nanoparquet and Arrow.
- `nanoparquet.write_data_page_version`: Data version to write by default. Possible values are 1 and 2. Default is 1.
- `nanoparquet.write_minmax_values`: Whether to write minimum and maximum values per row group, for data types that support this in `write_parquet()`.

License:

MIT

Author(s)

Maintainer: Gábor Csárdi <csardi.gabor@gmail.com>

Authors:

- Gábor Csárdi <csardi.gabor@gmail.com>
- Hannes Mühleisen (ORCID) [copyright holder]

Other contributors:

- Google Inc. [copyright holder]
- Apache Software Foundation [copyright holder]
- Posit Software, PBC [copyright holder]
- RAD Game Tools [copyright holder]
- Valve Software [copyright holder]
- Tenacious Software LLC [copyright holder]
- Facebook, Inc. [copyright holder]

See Also

Useful links:

- https://github.com/r-lib/nanoparquet
- https://nanoparquet.r-lib.org/
- Report bugs at https://github.com/r-lib/nanoparquet/issues

---

append_parquet *Append a data frame to an existing Parquet file*

Description

The schema of the data frame must be compatible with the schema of the file.

Usage

```
append_parquet(
  x,
  file,
  compression = c("snappy", "gzip", "zstd", "uncompressed"),
  encoding = NULL,
  row_groups = NULL,
  options = parquet_options()
)
```

Arguments

x	Data frame to append.
file	Path to the output file.
compression	Compression algorithm to use for the newly written data. See write_parquet().
encoding	Encoding to use for the newly written data. It does not have to be the same as the encoding of data in file. See write_parquet() for possible values.
row_groups	Row groups of the new, extended Parquet file. append_parquet() can only change the last existing row group, and if row_groups is specified, it has respect this. I.e. if the existing file has n rows, and the last row group starts at k (k <= n), then the first row group in row_groups that refers to the new data must start at k or n+1. (It is simpler to specify num_rows_per_row_group in options, see parquet_options() instead of row_groups. Only use row_groups if you need complete control.)
options	Nanoparquet options, for the new data, see parquet_options(). The keep_row_groups option also affects whether append_parquet() overwrites existing row groups in file.

Warning

This function is **not** atomic! If it is interrupted, it may leave the file in a corrupt state. To work around this create a copy of the original file, append the new data to the copy, and then rename the new, extended file to the original one.

About row groups

A Parquet file may be partitioned into multiple row groups, and indeed most large Parquet files are. `append_parquet()` is only able to update the existing file along the row group boundaries. There are two possibilities:

- `append_parquet()` keeps all existing row groups in `file`, and creates new row groups for the new data. This mode can be forced by the `keep_row_groups` option in `options`, see `parquet_options()`.
- Alternatively, `write_parquet` will overwrite the *last* row group in file, with its existing contents plus the (beginning of) the new data. This mode makes more sense if the last row group is small, because many small row groups are inefficient.

By default `append_parquet` chooses between the two modes automatically, aiming to create row groups with at least `num_rows_per_row_group` (see `parquet_options()`) rows. You can customize this behavior with the `keep_row_groups` options and the `row_groups` argument.

See Also

`write_parquet()`.

`edit_parquet_metadata` *Edit the key-value metadata of a Parquet file*

Description

Modify the file-level key-value metadata of a Parquet file in place, without copying any row-group data. Individual entries can be added, updated or removed.

Usage

```
edit_parquet_metadata(file, metadata)
```

Arguments

`file`	Path to a Parquet file.
`metadata`	Key-value pairs to add, update or remove. Accepted forms: • A **named character vector** – each name is a key and each element is its new value. Set a value to `NA` to **remove** that key. • A **data frame with two columns** – the first column contains keys and the second contains values. An `NA` value removes that key. • `NULL` – removes **all** key-value metadata from the file. Keys that are not mentioned in `metadata` are left unchanged.

Value

NULL, invisibly. Called for its side effect.

Warning

This function is **not** atomic! If it is interrupted it may leave the file in a corrupt state. To work around this create a copy of the original file, edit the copy, and then rename the copy to the original.

See Also

read_parquet_metadata() to inspect metadata. write_parquet() for the metadata argument when writing. append_parquet() to append rows without copying.

Examples

```
tmp <- tempfile(fileext = ".parquet")
write_parquet(data.frame(x = 1:5), tmp)

# Add a new entry
edit_parquet_metadata(tmp, c(created_by_user = "Alice"))
read_parquet_metadata(tmp)$file_meta_data$key_value_metadata

# Update an existing entry
edit_parquet_metadata(tmp, c(created_by_user = "Bob"))
read_parquet_metadata(tmp)$file_meta_data$key_value_metadata

# Remove an entry
edit_parquet_metadata(tmp, c(created_by_user = NA_character_))
read_parquet_metadata(tmp)$file_meta_data$key_value_metadata
```

---

infer_parquet_schema *Infer Parquet schema of a data frame*

Description

Infer Parquet schema of a data frame

Usage

```
infer_parquet_schema(df, options = parquet_options())
```

Arguments

df	Data frame.
options	Return value of parquet_options(), may modify the R to Parquet type mappings.

Value

Data frame, the inferred schema. It has the same columns as the return value of `read_parquet_schema()`: `file_name`, `r_col`, `name`, `r_type`, `type`, `type_length`, `repetition_type`, `converted_type`, `logical_type`, `num_children`, `scale`, `precision`, `field_id`.

See Also

`read_parquet_schema()` to read the schema of a Parquet file, `parquet_schema()` to create a Parquet schema from scratch.

`nanoparquet-types` *nanoparquet's type maps*

Description

How nanoparquet maps R types to Parquet types.

R's data types

When writing out a data frame, nanoparquet maps R's data types to Parquet logical types. The following table is a summary of the mapping. For the details see below.

R type	Parquet type	Default	Notes
bit64::integer64	INT64	x	`NA_integer64_` marks missing values.
blob::blob	BYTE_ARRAY	x	Missing values are `NULL`.
"	FIXED_LEN_BYTE_ARRAY		All entries must have the same length. Missing values are `NULL`
character	STRING(BYTE_ARRAY)	x	I.e. STRSXP. Converted to UTF-8.
"	BYTE_ARRAY		
"	FIXED_LEN_BYTE_ARRAY		
"	ENUM		
"	UUID		
Date	DATE	x	
difftime	INT64	x	If not hms::hms. Arrow metadata marks it as Duration(NS).
factor	STRING	x	Arrow metadata marks it as a factor.
"	ENUM		
hms::hms	TIME(true, MILLIS)	x	Sub-milliseconds precision is lost.
integer	INT(32, true)	x	I.e. INTSXP.
"	INT64		
"	INT96		
"	DECIMAL(INT32)		
"	DECIMAL(INT64)		
"	INT(8, *)		
"	INT(16, *)		
"	INT(32, signed)		
list	LIST(INT32 elements)	x	List of integer vectors. `NULL` entries and `NA` elements are suppo
"	LIST(DOUBLE elements)	x	List of double vectors. `NULL` entries and `NA` elements are suppo

"	LIST(STRING elements)	x	List of character vectors. NULL entries and NA elements are supp
"	BYTE_ARRAY		Must be a list of raw vectors. Missing values are NULL.
"	FIXED_LEN_BYTE_ARRAY		Must be a list of raw vectors of the same length. Missing value
logical	BOOLEAN	x	I.e. LGLSXP.
numeric	DOUBLE	x	I.e. REALSXP.
"	INT96		
"	FLOAT		
"	DECIMAL(INT32)		
"	DECIMAL(INT64)		
"	INT(*, *)		
"	FLOAT16		
POSIXct	TIMESTAMP(true, MICROS)	x	Sub-microsecond precision is lost.

The non-default mappings can be selected via the `schema` argument. E.g. to write out a factor column called 'name' as `ENUM`, use

```
write_parquet(..., schema = parquet_schema(name = "ENUM"))
```

The detailed mapping rules are listed below, in order of preference. These rules will likely change until nanoparquet reaches version 1.0.0.

1. `bit64::integer64` objects (from the bit64 package) are written as `INT64`. nanoparquet handles any object that inherits the `integer64` class this way. `NA_integer64_` (i.e. `INT64_MIN`) marks missing values.
2. `blob::blob` objects (from the blob package) are written as `BYTE_ARRAY`. `blob::blob` is a list of raw vectors, and nanoparquet handles any object that inherits the `blob` class this way, even if the blob package is not installed. Missing values (i.e. `NULL` list entries) are supported.
3. Factors (i.e. vectors that inherit the *factor* class) are converted to character vectors using `as.character()`, then written as a `STRSXP` (character vector) type. The fact that a column is a factor is stored in the Arrow metadata (see below), unless the `nanoparquet.write_arrow_metadata` option is set to `FALSE`.
4. Dates (i.e. the `Date` class) is written as `DATE` logical type, which is an `INT32` type internally.
5. `hms` objects (from the hms package) are written as `TIME(true, MILLIS)`. logical type, which is internally the `INT32` Parquet type. Sub-milliseconds precision is lost.
6. `POSIXct` objects are written as `TIMESTAMP(true, MICROS)` logical type, which is internally the `INT64` Parquet type. Sub-microsecond precision is lost.
7. `difftime` objects (that are not `hms` objects, see above), are written as an `INT64` Parquet type, and noting in the Arrow metadata (see below) that this column has type `Duration` with `NANOSECONDS` unit.
8. Integer vectors (`INTSXP`) are written as `INT(32, true)` logical type, which corresponds to the `INT32` type.
9. Real vectors (`REALSXP`) are written as the `DOUBLE` type.
10. Character vectors (`STRSXP`) are written as the `STRING` logical type, which has the `BYTE_ARRAY` type. They are always converted to UTF-8 before writing.

11. Logical vectors (`LGLSXP`) are written as the `BOOLEAN` type.
12. Other vectors error currently.

You can use `infer_parquet_schema()` on a data frame to map R data types to Parquet data types.

To change the default R to Parquet mapping, use `parquet_schema()` and the `schema` argument of `write_parquet()`. Currently supported non-default mappings are:

- `integer` to `INT64`,
- `integer` to `INT96`,
- `double` to `INT96`,
- `double` to `FLOAT`,
- `character` to `BYTE_ARRAY`,
- `character` to `FIXED_LEN_BYTE_ARRAY`,
- `character` to `ENUM`,
- `factor` to `ENUM`,
- `integer` to `DECIAML` & `INT32`,
- `integer` to `DECIAML` & `INT64`,
- `double` to `DECIAML` & `INT32`,
- `double` to `DECIAML` & `INT64`,
- `integer` to `INT(8, *)`, `INT(16, *)`, `INT(32, signed)`,
- `double` to `INT(*, *)`,
- `character` to `UUID`,
- `double` to `FLOAT16`,
- `list` of `integer` vectors to `LIST` with `INT32` elements,
- `list` of `double` vectors to `LIST` with `DOUBLE` elements,
- `list` of `character` vectors to `LIST` with `STRING` elements,
- `list` of `raw` vectors to `BYTE_ARRAY`,
- `list` of `raw` vectors to `FIXED_LEN_BYTE_ARRAY`,
- `blob::blob` to `FIXED_LEN_BYTE_ARRAY`.

Parquet's data types

When reading a Parquet file nanoparquet also relies on logical types and the Arrow metadata (if present, see below) in addition to the low level data types. The following table summarizes the mappings. See more details below.

Parquet type	R type	Notes
Logical types		
BSON	character	
DATE	Date	
DECIMAL	numeric	REALSXP, potentially losing precision.
ENUM	character	

FLOAT16	numeric	REALSXP
INT(8, *)	integer	
INT(16, *)	integer	
INT(32, *)	integer	Large unsigned values may overflow!
INT(64, *)	numeric	REALSXP, or `integer64` if `read_int64_type` option is set.
INTERVAL	list(raw)	Missing values are `NULL`.
JSON	character	
LIST	list	Elements are read as their corresponding R type.
MAP		Not supported.
STRING	factor	If Arrow metadata says it is a factor. Also UTF8.
"	character	Otherwise. Also UTF8.
TIME	hms::hms	Also TIME_MILLIS and TIME_MICROS.
TIMESTAMP	POSIXct	Also TIMESTAMP_MILLIS and TIMESTAMP_MICROS.
UUID	character	In `00112233-4455-6677-8899-aabbccddeeff` form.
UNKNOWN		Not supported.
Primitive types		
BOOLEAN	logical	
BYTE_ARRAY	factor	If Arrow metadata says it is a factor.
"	blob::blob	Otherwise. Missing values are `NULL`.
DOUBLE	numeric	REALSXP
FIXED_LEN_BYTE_ARRAY	blob::blob	Missing values are `NULL`.
FLOAT	numeric	REALSXP
INT32	integer	
INT64	numeric	REALSXP, or `integer64` if `read_int64_type` option is set.
INT96	POSIXct	

The exact rules are below. These rules will likely change until nanoparquet reaches version 1.0.0.

1. The `BOOLEAN` type is read as a logical vector (`LGLSXP`).
2. The `STRING` logical type and the `UTF8` converted type is read as a character vector with UTF-8 encoding.
3. The `DATE` logical type and the `DATE` converted type are read as a `Date` R object.
4. The `TIME` logical type and the `TIME_MILLIS` and `TIME_MICROS` converted types are read as an `hms` object, see the hms package.
5. The `TIMESTAMP` logical type and the `TIMESTAMP_MILLIS` and `TIMESTAMP_MICROS` converted types are read as `POSIXct` objects. If the logical type has the `UTC` flag set, then the time zone of the `POSIXct` object is set to `UTC`.
6. `INT32` is read as an integer vector (`INTSXP`).
7. `INT64` is read as a real vector (`REALSXP`) by default. If the `read_int64_type` option in `parquet_options()` is set to `"integer64"` or `"bit64::integer64"`, it is read as a `bit64::integer64` vector instead. `NA_integer64_` (i.e. `INT64_MIN`) marks missing values.
8. `DOUBLE` and `FLOAT` are read as real vectors (`REALSXP`).
9. `INT96` is read as a `POSIXct` read vector with the `tzone` attribute set to `"UTC"`. It was an old convention to store time stamps as `INT96` objects.

10. The DECIMAL converted type (FIXED_LEN_BYTE_ARRAY or BYTE_ARRAY type) is read as a real vector (REALSXP), potentially losing precision.
11. The ENUM logical type is read as a character vector.
12. The UUID logical type is read as a character vector that uses the 00112233-4455-6677-8899-aabbccddeeff form.
13. The FLOAT16 logical type is read as a real vector (REALSXP).
14. BYTE_ARRAY is read as a *factor* object if the file was written by Arrow and the original data type of the column was a factor. (See 'The Arrow metadata below.)
15. Otherwise BYTE_ARRAY and FIXED_LEN_BYTE_ARRAY are read as a blob::blob object. blob::blob is a list of raw vectors with class blob (and related vctrs classes). The blob package is not required to use this object; it is simply a list of raw vectors. Missing values are denoted by NULL.

Other logical and converted types are read as their annotated low level types:

1. INT(8, true), INT(16, true) and INT(32, true) are read as integer vectors because they are INT32 internally in Parquet.
2. INT(64, true) is read as a real vector (REALSXP), unless the read_int64_type option is set (see above).
3. Unsigned integer types INT(8, false), INT(16, false) and INT(32, false) are read as integer vectors (INTSXP). Large positive values may overflow into negative values, this is a known issue that we will fix.
4. INT(64, false) is read as a real vector (REALSXP), unless the read_int64_type option is set (see above). Large positive values may overflow into negative values, this is a known issue that we will fix.
5. INTERVAL is a fixed length byte array, and nanoparquet reads it as a list of raw vectors. Missing values are denoted by NULL.
6. JSON columns are read as character vectors (STRSXP).
7. BSON columns are read as raw vectors (RAWSXP).

These types are not yet supported:

1. Nested LIST types (lists of lists) are not supported.
2. The MAP logical type is not supported.
3. The UNKNOWN logical type is not supported.

You can use the read_parquet_schema() function to see how R would read the columns of a Parquet file. Look at the r_type column.

The Arrow metadata

Apache Arrow (i.e. the arrow R package) adds additional metadata to Parquet files when writing them in arrow::write_parquet(). Then, when reading the file in arrow::read_parquet(), it uses this metadata to recreate the same Arrow and R data types as before writing.

nanoparquet::write_parquet() also adds the Arrow metadata to Parquet files, unless the nanoparquet.write_arrow_me
option is set to FALSE.

Similarly, `nanoparquet::read_parquet()` uses the Arrow metadata in the Parquet file (if present), unless the `nanoparquet.use_arrow_metadata` option is set to FALSE.

The Arrow metadata is stored in the file level key-value metadata, with key `ARROW:schema`.

Currently nanoparquet uses the Arrow metadata for two things:

- It uses it to detect factors. Without the Arrow metadata factors are read as string vectors.
- It uses it to detect `difftime` objects. Without the arrow metadata these are read as `INT64` columns, containing the time difference in nanoseconds.

See Also

nanoparquet-package for options that modify the type mappings.

`parquet-encodings` *Parquet encodings*

Description

Various Parquet encodings

Nanoparquet defaults

Currently the defaults are decided based on the R types. This might change in the future. In general, the defaults will likely change until nanoparquet reaches version 1.0.0.

Current encoding defaults:

- Definition levels always use `RLE`. (Nanoparquet does not currently write repetition levels, but they'll also use `RLE`, once implemented.)
- `factor` columns use `RLE_DICTIONARY`.
- `logical` columns use `RLE` if the average run length of the first 10,000 values is at least 15. Otherwise they use the `PLAIN` encoding.
- `integer`, `double` and `character` columns use `RLE_DICTIONARY` if at least two third of their values are repeated. Otherwise they use `PLAIN` encoding.
- `list` columns of `raw` vectors always use the `PLAIN` encoding currently.

Parquet encodings

See https://github.com/apache/parquet-format/blob/master/Encodings.md for more details on Parquet encodings.

`PLAIN` **encoding:**

Supported types: all.

In general values are written back to back:

- Integer types are little endian.
- Floating point types follow the IEEE standard.

- BYTE_ARRAY: for each element, there is a little endian 4-byte length and then the bytes themselves.
- FIXED_LEN_BYTE_ARRAY: bytes are written back to back.

Nanoparquet can read and write this encoding for all primitive types.

RLE_DICTIONARY **encoding:**

Supported types: dictionary indices in data pages.

This encoding combines run-length encoding and bit-packing. Repeated sequences of the same value can be run-length encoded, and non-repeated parts are bit packed. It is used for data pages of dictionaries. The dictionary pages themselves are PLAIN encoded.

The deprecated PLAIN_DICTIONARY name is treated the same as RLE_DICTIONARY.

Nanoparquet can read and write this encoding.

RLE **encoding:**

Supported types: BOOLEAN. Also for definition and repetition levels.

This is the same encoding as RLE_DICTIONARY, with a slightly different header. It combines run-length encoding and bit packing. It is used for BOOLEAN columns, and also for definition and repetition levels.

Nanoparquet can read and write this encoding.

BIT_PACKED **encoding (deprecated in favor of** RLE**):**

Supported types: none. Only for definition and repetition levels, but RLE should be used instead.

This is a simple bit packing encoding for integers, that was previously used for encoding definition and repetition levels. It is not used in new Parquet files because the the RLE encoding includes it and it is better.

Nanoparquet currently cannot read or write the BIT_PACKED encoding.

DELTA_BINARY_PACKED **encoding:**

Supported types: INT32, INT64.

This encoding efficiently encodes integer columns if the differences between consecutive elements are often the same, and/or the differences between consecutive elements are small. The extreme case of an arithmetic sequence can be encoded in O(1) space.

Nanoparquet can read this encoding, but cannot currently write it.

DELTA_LENGTH_BYTE_ARRAY **encoding:**

Supported types: BYTE_ARRAY.

This encoding uses DELTA_BINARY_PACKED to encode the length of all byte array elements. It is especially efficient for short byte array elements, i.e. a column of short strings.

Nanoparquet can read this encoding, but cannot currently write it.

DELTA_BYTE_ARRAY **encoding:**

Supported types: BYTE_ARRAY, FIXED_LEN_BYTE_ARRAY.

This encoding is efficient if consecutive byte array elements share the same prefix, because each element can reuse a prefix of the previous element.

Nanoparquet can read this encoding, but cannot currently write it.

`BYTE_STREAM_SPLIT` **encoding:**

Supported types: `FLOAT`, `DOUBLE`, `INT32`, `INT64`, `FIXED_LEN_BYTE_ARRAY`.

This encoding stores the first bytes of the elements first, then the second bytes, etc. It does not reduce the size in itself, but may allow more efficient compression.

Nanoparquet can read this encoding, but cannot currently write it.

See Also

`write_parquet()` on how to select a non-default encoding when writing Parquet files.

---

parquet_column_types *Map between R and Parquet data types*

Description

Note that this function is now deprecated. Please use `read_parquet_schema()` for files, and `infer_parquet_schema()` for data frames.

Usage

```
parquet_column_types(x, options = parquet_options())
```

Arguments

`x`	Path to a Parquet file, or a data frame.
`options`	Nanoparquet options, see `parquet_options()`.

Details

This function works two ways. It can map the R types of a data frame to Parquet types, to see how `write_parquet()` would write out the data frame. It can also map the types of a Parquet file to R types, to see how `read_parquet()` would read the file into R.

Value

Data frame with columns:

- `file_name`: file name.
- `name`: column name.
- `type`: (low level) Parquet data type.
- `r_type`: the R type that corresponds to the Parquet type. Might be `NA` if `read_parquet()` cannot read this column. See nanoparquet-types for the type mapping rules.
- `repetition_type`: whether the column in `REQUIRED` (cannot be `NA`) or `OPTIONAL` (may be `NA`). `REPEATED` columns are not currently supported by nanoparquet.
- `logical_type`: Parquet logical type in a list column. An element has at least an entry called `type`, and potentially additional entries, e.g. `bit_width`, `is_signed`, etc.

See Also

`read_parquet_metadata()` to read more metadata, `read_parquet_info()` for a very short summary. `read_parquet_schema()` for the complete Parquet schema. `read_parquet()`, `write_parquet()`, nanoparquet-types.

`parquet_options` *Nanoparquet options*

Description

Create a list of nanoparquet options.

Usage

```
parquet_options(
  class = getOption("nanoparquet.class", "tbl"),
  compression_level = getOption("nanoparquet.compression_level", NA_integer_),
  read_int64_type = getOption("nanoparquet.read_int64_type", "double"),
  keep_row_groups = FALSE,
 num_rows_per_row_group = getOption("nanoparquet.num_rows_per_row_group", 122880L),
  use_arrow_metadata = getOption("nanoparquet.use_arrow_metadata", TRUE),
  write_arrow_metadata = getOption("nanoparquet.write_arrow_metadata", TRUE),
 write_data_page_version = getOption("nanoparquet.write_data_page_version", 1L),
  write_minmax_values = getOption("nanoparquet.write_minmax_values", TRUE)
)
```

Arguments

`class` The extra class or classes to add to data frames created in `read_parquet()`. By default nanoparquet adds the `"tbl"` class, so data frames are printed differently if the pillar package is loaded.

`compression_level` The compression level in `write_parquet()`. `NA` is the default, and it specifies the default compression level of each method. `Inf` always selects the highest possible compression level. More details:

- Snappy does not support compression levels currently.
- GZIP supports levels from 0 (uncompressed), 1 (fastest), to 9 (best). The default is 6.
- ZSTD allows positive levels up to 22 currently. 20 and above require more memory. Negative levels are also allowed, the lower the level, the faster the speed, at the cost of compression. Currently the smallest level is -131072. The default level is 3.

`read_int64_type` How to represent INT64 columns with a 64-bit integer logical type in `read_parquet()` and `read_parquet_schema()`. Possible values:

- "double" (the default): read as a regular R double vector, which may lose precision for large values.
- "integer64" or "bit64::integer64": read as a bit64::integer64 vector. Requires the bit64 package to be installed.

keep_row_groups
: This option is used when appending to a Parquet file with append_parquet(). If TRUE then the existing row groups of the file are always kept as is and nanoparquet creates new row groups for the new data. If FALSE (the default), then the last row group of the file will be overwritten if it is smaller than the default row group size, i.e. num_rows_per_row_group.

num_rows_per_row_group
: The number of rows to put into a row group, if row groups are not specified explicitly. It should be an integer scalar. Defaults to 122,880, the same as DuckDB's default, so files written by nanoparquet work well with DuckDB, Arrow and other tools that read row groups in parallel.

use_arrow_metadata
: TRUE or FALSE. If TRUE, then read_parquet() and read_parquet_schema() will make use of the Apache Arrow metadata to assign R classes to Parquet columns. This is currently used to detect factor columns, and to detect "difftime" columns.

  If this option is FALSE:

  - "factor" columns are read as character vectors.
  - "difftime" columns are read as real numbers, meaning one of seconds, milliseconds, microseconds or nanoseconds. Impossible to tell which without using the Arrow metadata.

write_arrow_metadata
: Whether to add the Apache Arrow types as metadata to the file write_parquet().

write_data_page_version
: Data version to write by default. Possible values are 1 and 2. Default is 1.

write_minmax_values
: Whether to write minimum and maximum values per row group, for data types that support this in write_parquet(). However, nanoparquet currently does not support minimum and maximum values for the DECIMAL, UUID and FLOAT16 logical types and the BOOLEAN, BYTE_ARRAY and FIXED_LEN_BYTE_ARRAY primitive types if they are writing without a logical type. Currently the default is TRUE.

Value

List of nanoparquet options.

Examples

```
# the effect of using Arrow metadata
tmp <- tempfile(fileext = ".parquet")
d <- data.frame(
  fct = as.factor("a"),
  dft = as.difftime(10, units = "secs")
```

```
)
write_parquet(d, tmp)
read_parquet(tmp, options = parquet_options(use_arrow_metadata = TRUE))
read_parquet(tmp, options = parquet_options(use_arrow_metadata = FALSE))
```

`parquet_schema` *Create a Parquet schema*

Description

You can use this schema to specify how to write out a data frame to a Parquet file with `write_parquet()`.

Usage

```
parquet_schema(...)
```

Arguments

`...`	Parquet type specifications, see below. For backwards compatibility, you can supply a file name here, and then `parquet_schema` behaves as `read_parquet_schema()`.

Details

A schema is a list of potentially named type specifications. A schema is stored in a data frame. Each (potentially named) argument of `parquet_schema` may be a character scalar, or a list. Parameterized types need to be specified as a list. Primitive Parquet types may be specified as a string or a list.

Value

Data frame with the same columns as `read_parquet_schema()`: `file_name`, `r_col`, `name`, `r_type`, `type`, `type_length`, `repetition_type`, `converted_type`, `logical_type`, `num_children`, `scale`, `precision`, `field_id`.

Possible types:

Special type:

- `"AUTO"`: this is not a Parquet type, but it tells `write_parquet()` to map the R type to Parquet automatically, using the default mapping rules.

Primitive Parquet types:

- `"BOOLEAN"`
- `"INT32"`
- `"INT64"`
- `"INT96"`

- "FLOAT"
- "DOUBLE"
- "BYTE_ARRAY"
- "FIXED_LEN_BYTE_ARRAY": fixed-length byte array. It needs a type_length parameter, an integer between 0 and 2^31-1.

Parquet logical types:

- "STRING"
- "ENUM"
- "UUID"
- "INTEGER": signed or unsigned integer. It needs a bit_width and an is_signed parameter. bit_width must be 8, 16, 32 or 64. is_signed must be TRUE or FALSE.
- "INT": same as "INTEGER". The Parquet documentation uses "INT", but the actual specification uses "INTEGER". Both are supported in nanoparquet.
- "DECIMAL": decimal number of specified scale and precision. It needs the precision and primitive_type parameters. Also supports the scale parameter, it defaults to zero if not specified.
- "FLOAT16"
- "DATE"
- "TIME": needs an is_adjusted_utc (TRUE or FALSE) and a unit parameter. unit must be "MILLIS", "MICROS" or "NANOS".
- "TIMESTAMP": needs an is_adjusted_utc (TRUE or FALSE) and a unit parameter. unit must be "MILLIS", "MICROS" or "NANOS".
- "JSON"
- "BSON"
- "LIST": list of some other type. It needs an element parameter, which is a type specification for the list elements. Currently element can be only "INT32", "DOUBLE" or "STRING". Also, currently nanoparquet always write LIST columns that are "OPTIONAL", both for the list elements and the list itself.

Logical types MAP, and UNKNOWN are not supported currently.

Converted types are deprecated in the Parquet specification in favor of logical types, but parquet_schema() accepts some converted types as a syntactic shortcut for the corresponding logical types:

- INT_8 mean list("INT", bit_width = 8, is_signed = TRUE).
- INT_16 mean list("INT", bit_width = 16, is_signed = TRUE).
- INT_32 mean list("INT", bit_width = 32, is_signed = TRUE).
- INT_64 mean list("INT", bit_width = 64, is_signed = TRUE).
- TIME_MICROS means list("TIME", is_adjusted_utc = TRUE, unit = "MICROS").
- TIME_MILLIS means list("TIME", is_adjusted_utc = TRUE, unit = "MILLIS").
- TIMESTAMP_MICROS means list("TIMESTAMP", is_adjusted_utc = TRUE, unit = "MICROS").
- TIMESTAMP_MILLIS means list("TIMESTAMP", is_adjusted_utc = TRUE, unit = "MILLIS").

- UINT_8 means `list("INT", bit_width = 8, is_signed = FALSE)`.
- UINT_16 means `list("INT", bit_width = 16, is_signed = FALSE)`.
- UINT_32 means `list("INT", bit_width = 32, is_signed = FALSE)`.
- UINT_64 means `list("INT", bit_width = 64, is_signed = FALSE)`.

Missing values:

Each type might also have a `repetition_type` parameter, with possible values `"REQUIRED"`, `"OPTIONAL"` or `"REPEATED"`. `"REQUIRED"` columns do not allow missing values. Missing values are allowed in `"OPTIONAL"` columns. `"REPEATED"` columns are currently not supported in `write_parquet()`.

Examples

```
parquet_schema(
  c1 = "INT32",
  c2 = list("INT", bit_width = 64, is_signed = TRUE),
  c3 = list("STRING", repetition_type = "OPTIONAL"),
  l = list("LIST", element = "DOUBLE")
)
```

---

read_parquet *Read a Parquet file into a data frame*

Description

Converts the contents of the named Parquet file to a R data frame.

Usage

```
read_parquet(file, col_select = NULL, options = parquet_options())
```

Arguments

`file`	Path to a Parquet file. It may also be an R connection, in which case it first reads all data from the connection, writes it into a temporary file, then reads the temporary file, and deletes it. The connection might be open, it which case it must be a binary connection. If it is not open, then `read_parquet()` will open it and also close it in the end.
`col_select`	Columns to read. It can be a numeric vector of column indices, or a character vector of column names. It is an error to select the same column multiple times. The order of the columns in the result is the same as the order in `col_select`.
`options`	Nanoparquet options, see `parquet_options()`.

Value

A `data.frame` with the file's contents.

See Also

See write_parquet() to write Parquet files, nanoparquet-types for the R <-> Parquet type mapping. See read_parquet_info(), for general information, read_parquet_schema() for information about the columns, and read_parquet_metadata() for the complete metadata.

Examples

```
file_name <- system.file("extdata/userdata1.parquet", package = "nanoparquet")
parquet_df <- nanoparquet::read_parquet(file_name)
print(str(parquet_df))
```

---

read_parquet_info *Short summary of a Parquet file*

Description

Short summary of a Parquet file

Usage

```
read_parquet_info(file)

parquet_info(file)
```

Arguments

file	Path to a Parquet file.

Value

Data frame with columns:

- file_name: file name.
- num_cols: number of columns. (The number of child nodes of the root node in the schema.)
- num_rows: number of rows.
- num_row_groups: number of row groups.
- file_size: file size in bytes.
- parquet_version: Parquet version.
- created_by: A string scalar, usually the name of the software that created the file. NA if not available.

See Also

read_parquet_metadata() to read more metadata, read_parquet_schema() for column information. read_parquet(), write_parquet(), nanoparquet-types.

`read_parquet_metadata` *Read the metadata of a Parquet file*

Description

This function should work on all files, even if `read_parquet()` is unable to read them, because of an unsupported schema, encoding, compression or other reason.

Usage

```
read_parquet_metadata(file, options = parquet_options())

parquet_metadata(file)
```

Arguments

`file`	Path to a Parquet file.
`options`	Options that potentially alter the default Parquet to R type mappings, see `parquet_options()`.

Value

A named list with entries:

- `file_meta_data`: a data frame with file meta data:
  - `file_name`: file name.
  - `version`: Parquet version, an integer.
  - `num_rows`: total number of rows.
  - `key_value_metadata`: list column of a data frames with two character columns called `key` and `value`. This is the key-value metadata of the file. Arrow stores its schema here.
  - `created_by`: A string scalar, usually the name of the software that created the file.
- `schema`: data frame, the schema of the file. It has one row for each node (inner node or leaf node). For flat files this means one root node (inner node), always the first one, and then one row for each "real" column. For nested schemas, the rows are in depth-first search order. Most important columns are:
  - `file_name`: file name.
  - `r_col`: integer, the column index in R, starting from one. `NA` for the root node. For Parquet files without list columns this is simply the column index in the file. A list column has multiple schema rows, and they all have the same `r_col`.
  - `name`: column name.
  - `r_type`: the R type that corresponds to the Parquet type. Might be `NA` if `read_parquet()` cannot read this column. See nanoparquet-types for the type mapping rules.
  - `r_type`:
  - `type`: data type. One of the low level data types.
  - `type_length`: length for fixed length byte arrays.

  - repettion_type: character, one of REQUIRED, OPTIONAL or REPEATED.
  - logical_type: a list column, the logical types of the columns. An element has at least an entry called type, and potentially additional entries, e.g. bit_width, is_signed, etc.
  - num_children: number of child nodes. Should be a non-negative integer for the root node, and NA for a leaf node.
- $row_groups: a data frame, information about the row groups. Some important columns:
  - file_name: file name.
  - id: row group id, integer from zero to number of row groups minus one.
  - total_byte_size: total uncompressed size of all column data.
  - num_rows: number of rows.
  - file_offset: where the row group starts in the file. This is optional, so it might be NA.
  - total_compressed_size: total byte size of all compressed (and potentially encrypted) column data in this row group. This is optional, so it might be NA.
  - ordinal: ordinal position of the row group in the file, starting from zero. This is optional, so it might be NA. If NA, then the order of the row groups is as they appear in the metadata.
- $column_chunks: a data frame, information about all column chunks, across all row groups. Some important columns:
  - file_name: file name.
  - row_group: which row group this chunk belongs to.
  - column: which leaf column this chunks belongs to. The order is the same as in $schema, but only leaf columns (i.e. columns with NA children) are counted.
  - file_path: which file the chunk is stored in. NA means the same file.
  - file_offset: where the column chunk begins in the file.
  - type: low level parquet data type.
  - encodings: encodings used to store this chunk. It is a list column of character vectors of encoding names. Current possible encodings: "PLAIN", "GROUP_VAR_INT", "PLAIN_DICTIONARY", "RLE", "BIT_PACKED", "DELTA_BINARY_PACKED", "DELTA_LENGTH_BYTE_ "DELTA_BYTE_ARRAY", "RLE_DICTIONARY", "BYTE_STREAM_SPLIT".
  - path_in_scema: list column of character vectors. It is simply the path from the root node. It is simply the column name for flat schemas.
  - codec: compression codec used for the column chunk. Possible values are: "UNCOMPRESSED", "SNAPPY", "GZIP", "LZO", "BROTLI", "LZ4", "ZSTD".
  - num_values: number of values in this column chunk.
  - total_uncompressed_size: total uncompressed size in bytes.
  - total_compressed_size: total compressed size in bytes.
  - data_page_offset: absolute position of the first data page of the column chunk in the file.
  - index_page_offset: absolute position of the first index page of the column chunk in the file, or NA if there are no index pages.
  - dictionary_page_offset: absolute position of the first dictionary page of the column chunk in the file, or NA if there are no dictionary pages.
  - null_count: the number of missing values in the column chunk. It may be NA.
  - min_value: list column of raw vectors, the minimum value of the column, in binary. If NULL, then then it is not specified. This column is experimental.

- `max_value`: list column of raw vectors, the maximum value of the column, in binary. If `NULL`, then then it is not specified. This column is experimental.
- `is_min_value_exact`: whether the minimum value is an actual value of a column, or a bound. It may be `NA`.
- `is_max_value_exact`: whether the maximum value is an actual value of a column, or a bound. It may be `NA`.

See Also

`read_parquet_info()` for a much shorter summary. `read_parquet_schema()` for column information. `read_parquet()` to read, `write_parquet()` to write Parquet files, nanoparquet-types for the R <-> Parquet type mappings.

Examples

```
file_name <- system.file("extdata/userdata1.parquet", package = "nanoparquet")
nanoparquet::read_parquet_metadata(file_name)
```

---

read_parquet_schema *Read the schema of a Parquet file*

Description

This function should work on all files, even if `read_parquet()` is unable to read them, because of an unsupported schema, encoding, compression or other reason.

Usage

```
read_parquet_schema(file, options = parquet_options())
```

Arguments

`file`	Path to a Parquet file.
`options`	Return value of `parquet_options()`, options that potentially modify the Parquet to R type mappings.

Value

```
Data frame, the schema of the file. It has one row for
each node (inner node or leaf node). For flat files this means one
root node (inner node), always the first one, and then one row for
each "real" column. For nested schemas, the rows are in depth-first
search order. Most important columns are:
- `file_name`: file name.
- `r_col`: integer, the column index in R, starting from one.
  `NA` for the root node. For Parquet files without list columns
  this is simply the column index in the file. A list column has
```

```
  multiple schema rows, and they all have the same `r_col`.
- `name`: column name.
- `r_type`: the R type that corresponds to the Parquet type.
  Might be `NA` if [read_parquet()] cannot read this column. See
  [nanoparquet-types] for the type mapping rules.
- `type`: data type. One of the low level data types.
- `type_length`: length for fixed length byte arrays.
- `repettion_type`: character, one of `REQUIRED`, `OPTIONAL` or
  `REPEATED`.
- `logical_type`: a list column, the logical types of the columns.
  An element has at least an entry called `type`, and potentially
  additional entries, e.g. `bit_width`, `is_signed`, etc.
- `num_children`: number of child nodes. Should be a non-negative
  integer for the root node, and `NA` for a leaf node.
```

See Also

`read_parquet_metadata()` to read more metadata, `read_parquet_info()` to show only basic information. `read_parquet()`, `write_parquet()`, nanoparquet-types.

`write_parquet` *Write a data frame to a Parquet file*

Description

Writes the contents of an R data frame into a Parquet file.

Usage

```
write_parquet(
  x,
  file,
  schema = NULL,
  compression = c("snappy", "gzip", "zstd", "uncompressed"),
  encoding = NULL,
  metadata = NULL,
  row_groups = NULL,
  options = parquet_options()
)
```

Arguments

`x`	Data frame to write.
`file`	Path to the output file. If this is the string `":raw:"`, then the data frame is written to a memory buffer, and the memory buffer is returned as a raw vector. If `":stdout:"`, then it is written to the standard output. (When writing to the standard output, special care is needed to make sure no regular R output gets mixed up with the Parquet bytes!)

schema — Parquet schema. Specify a schema to tweak the default nanoparquet R -> Parquet type mappings. Use `parquet_schema()` to create a schema that you can use here, or `read_parquet_schema()` to use the schema of a Parquet file.

compression — Compression algorithm to use. Currently `"snappy"` (the default), `"gzip"`, `"zstd"`, and `"uncompressed"` are supported.

encoding — Encoding to use. Possible values:

- If `NULL`, the appropriate encoding is selected automatically: `RLE` or `PLAIN` for `BOOLEAN` columns, `RLE_DICTIONARY` for other columns with many repeated values, and `PLAIN` otherwise.
- If It is a single (unnamed) character string, then it'll be used for all columns.
- If it is an unnamed character vector of encoding names of the same length as the number of columns in the data frame, then those encodings will be used for each column.
- If it is a named character vector, then the named must be unique and each name must match a column name, to specify the encoding of that column. The special empty name (`""`) applies to the rest of the columns. If there is no empty name, the rest of the columns will use the default encoding.

If `NA_character_` is specified for a column, the default encoding is used for the column.

If a specified encoding is invalid for a certain column type, or nanoparquet does not implement it, `write_parquet()` throws an error.

Currently `write_parquet()` supports the following encodings:

- `PLAIN` for all column types,
- `PLAIN_DICTIONARY` and `RLE_DICTIONARY` for all column types,
- `RLE` for BOOLEAN columns.

See parquet-encodings for more about encodings.

metadata — Additional key-value metadata to add to the file. This must be a named character vector, or a data frame with columns character columns called `key` and `value`.

row_groups — Row groups of the Parquet file. If `NULL`, then the `num_rows_per_row_group` option is used from the `options` argument, see `parquet_options()`. Otherwise it must be an integer vector, specifying the starts of the row groups.

options — Nanoparquet options, see `parquet_options()`.

Details

`write_parquet()` converts string columns to UTF-8 encoding by calling `base::enc2utf8()`. It does the same for factor levels.

Value

`NULL`, unless `file` is `":raw:"`, in which case the Parquet file is returned as a raw vector.

See Also

`read_parquet_metadata()`, `read_parquet()`.

Examples

```
# add row names as a column, because `write_parquet()` ignores them.
mtcars2 <- cbind(name = rownames(mtcars), mtcars)
write_parquet(mtcars2, "mtcars.parquet")
```

Index

append_parquet, 5
append_parquet(), *5*, *7*, *17*

base::enc2utf8(), *26*

edit_parquet_metadata, 6

infer_parquet_schema, 7
infer_parquet_schema(), *10*, *15*

nanoparquet *(*nanoparquet-package*)*, 2
nanoparquet-package, 2, *13*
nanoparquet-types, 8, *15, 16*, *21, 22*, *24, 25*

parquet-encodings, 13, *26*
parquet_column_types, 15
parquet_info *(*read_parquet_info*)*, 21
parquet_metadata
*(*read_parquet_metadata*)*, 22
parquet_options, 16
parquet_options(), *5–7*, *15*, *20*, *22*, *24*, *26*
parquet_schema, 18
parquet_schema(), *8*, *10*, *26*

read_parquet, 20
read_parquet(), *15–17*, *21, 22*, *24–26*
read_parquet_info, 21
read_parquet_info(), *16*, *21*, *24, 25*
read_parquet_metadata, 22
read_parquet_metadata(), *7*, *16*, *21*, *25, 26*
read_parquet_schema, 24
read_parquet_schema(), *8*, *12*, *15–18*, *21*,
24, *26*

write_parquet, 25
write_parquet(), *5–7*, *10*, *15–18*, *20, 21*, *24,*
25