Package: qio (via r-universe)

October 29, 2026

Title Read and Write 'Apache Parquet' Files

Version 0.1.1

Description Read and write 'Apache Parquet' files. Whole files are read with a single call, and larger ones can be opened to inspect their schema and read selected columns, row groups, or batches. Built on the bundled C library 'carquet', with no required R package dependencies.

URL https://pedrobtz.github.io/qio/, https://github.com/pedrobtz/qio

BugReports https://github.com/pedrobtz/qio/issues

License MIT + file LICENSE

Copyright file inst/COPYRIGHTS

Encoding UTF-8

SystemRequirements GNU make, zlib

RoxygenNote 8.0.0

Depends R (>= 3.5.0)

Imports utils

Suggests bit64, dplyr, hms, testthat (>= 3.0.0), withr

Config/testthat/edition 3

Config/Needs/website knitr, rmarkdown

NeedsCompilation yes

Author Pedro Baltazar [aut, cre, cph], Johan HG Natter [aut, cph] (carquet source code (MIT license)), vctrs authors [ctb, cph] (qio_s3_register() helper (MIT license))

Maintainer Pedro Baltazar <pedrobtz@gmail.com>

Config/pak/sysreqs make zlib1g-dev

Repository https://cran.r-universe.dev

Date/Publication 2026-09-29 14:20:17 UTC

RemoteUrl https://github.com/cran/qio

RemoteRef HEAD

RemoteSha 322dcdd9562cff4654c5d44b6128d8d4965f4456

Contents

bloom_filter_may_contain . . . 2
close_parquet . . . 3
collect . . . 4
column_chunks . . . 6
column_statistics . . . 7
infer_parquet_schema . . . 9
metadata . . . 9
open_parquet . . . 10
page_index . . . 11
parquet_schema . . . 12
parquet_type_mapping . . . 13
qio-limitations . . . 14
qio-types . . . 15
read_parquet . . . 17
read_plan . . . 18
row_groups . . . 21
schema . . . 22
validate_parquet . . . 23
walk_batches . . . 24
write_parquet . . . 26

Index **28**

`bloom_filter_may_contain` *Test values against a Parquet bloom filter*

Description

A bloom filter answers one question: is this value *definitely absent* from the column chunk? It never proves presence. `FALSE` means the value is not there; `TRUE` means it may be, and only reading can settle it.

Usage

```
bloom_filter_may_contain(x, column, values, row_group = 1L, ...)

## S3 method for class 'qio_parquet_file'
bloom_filter_may_contain(x, column, values, row_group = 1L, ...)
```

Arguments

`x`	A `qio_parquet_file` object.
`column`	A single complete column path, as `schema()` reports it in `path` and `names()` returns it – not the bare `name`, which is not unique across a nested file.

values	Values to test. Numeric for numeric columns, character for byte-array columns. NA returns NA.
row_group	Row group to test, 1-based. A bloom filter belongs to one column chunk, so it covers one row group.
...	Reserved for future use.

Details

Values are matched against the column's physical type, because that is what the writer hashed. A value qio cannot reduce to that type is an error rather than a FALSE, which would read as "definitely absent".

qio's own writer does not emit bloom filters; see qio-limitations. Use column_chunks() to find out whether a chunk has one.

Value

A logical vector the length of values: FALSE where the value is definitely absent, TRUE where it may be present.

See Also

column_chunks()

Examples

```
# qio's writer emits no bloom filters, so this uses a bundled file written
# by pyarrow. Its `key` column runs 0..3999 across four row groups.
path <- system.file("extdata", "bloom_sorted.parquet", package = "qio")
pf <- open_parquet(path)

# Row group 1 holds keys 0..999. A present value may be present; absent
# values are ruled out, and never wrongly, because there are no false
# negatives.
bloom_filter_may_contain(pf, "key", c(42, 123456))

# Which chunks even have a filter to test.
chunks <- column_chunks(pf)
chunks[chunks$row_group == 1, c("path", "bloom_filter")]

close_parquet(pf)
```

---

close_parquet *Close a Parquet file*

Description

Explicitly releases the native resources owned by an open Parquet handle. Closing an already closed handle has no effect.

Usage

```
close_parquet(x)
```

Arguments

x	A qio_parquet_file object.

Value

x, invisibly.

See Also

open_parquet()

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(mtcars, path)
pf <- open_parquet(path)
close_parquet(pf)
```

---

collect *Collect data from a Parquet file*

Description

Columns are returned in requested order. Row groups are always returned in their physical file order, even if their selector is not sorted.

Usage

```
collect(x, ...)

## S3 method for class 'qio_parquet_file'
collect(
  x,
  ...,
  columns = NULL,
  row_groups = NULL,
  batch_size = 65536L,
  int64 = c("double", "integer64"),
  time = c("numeric", "hms"),
  tz = "UTC",
  verbose = FALSE
)
```

Arguments

x	A qio_parquet_file object.
...	Reserved for future use.
columns	Character vector of complete column paths, or NULL for all columns. Paths are matched exactly as schema() and names() report them, dot-separated for nested leaves, and never by leaf name alone: two leaves may share a name under different parents. An unknown path is an error, and so is a path matching more than one leaf.
row_groups	Integer vector of 1-based row-group IDs, or NULL for all row groups.
batch_size	Positive number of rows decoded at a time. It bounds the reader's scratch memory, not the size of the result (see Details); walk_batches() additionally uses it as the size of each batch.
int64	How 64-bit integer columns reach R. "double" (the default) is exact from -2^53 through 2^53 and returns NA outside it. "integer64" returns bit64::integer64, which needs the suggested bit64 package and covers the signed 64-bit range **except its lowest value**: bit64 reserves -9223372036854775808 as its own NA, so a column storing INT64_MIN reads as NA in either mode. Either way values that cannot be represented become NA, and one warning naming the column is emitted for each column that lost values – once per column, however many values, row groups, or batches were affected, and never for a column that lost nothing. Unsigned 64-bit columns are never returned as negative numbers.
time	How TIME columns reach R: "numeric" (the default) returns seconds since midnight, "hms" returns hms::hms and needs the suggested hms package. Neither returns POSIXct, because a time of day is not an instant.
tz	Time zone name, "UTC" by default. A UTC-adjusted TIMESTAMP is an instant, so tz changes only how it prints. A non-UTC TIMESTAMP is a wall clock with no zone stored, so its civil components are interpreted in tz; base R decides ambiguous and nonexistent times at daylight-saving boundaries. The machine's local zone is never used implicitly.
verbose	Report what the read is about to do before doing it: the rows, columns, and row groups selected, the batch size, and the resolved read_plan() for the selected columns only – not for the whole file, so it answers "what am I about to get". Written with message(), so it goes to stderr and suppressMessages() silences it.

Details

collect() returns the whole selection, so batch_size does not bound the result; use walk_batches() for that. It does bound the scratch memory the reader allocates while decoding string and binary columns, which would otherwise scale with the largest selected row group rather than with anything the caller controls. Smaller batches lower peak memory and cost a little throughput on dictionary-encoded text.

Nested and repeated columns are not materialized in qio 0.1.0. When a selection includes them, they are omitted and one message reports how many physical leaf columns were skipped. Nested reading is deferred to qio 0.2.0. If every selected column is nested, the result is a zero-column data frame with the selected number of rows.

Value

A data frame.

See Also

`open_parquet()` for the handle and for `mmap` and `threads`, `walk_batches()` to process a file that does not fit in memory, `read_plan()` to preview the R type of every column, and `read_parquet()`, which is `open_parquet()` plus `collect()` for a whole file.

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(mtcars, path)
pf <- open_parquet(path)
collect(pf, columns = c("mpg", "cyl"))
close_parquet(pf)
```

`column_chunks` *Inspect Parquet column chunks*

Description

Reports how each column is stored in each row group: its physical type, compression, sizes, encodings, and which optional structures are present. One row per column per row group.

Usage

```
column_chunks(x, ...)

## S3 method for class 'qio_parquet_file'
column_chunks(x, ...)
```

Arguments

`x`	A `qio_parquet_file` object.
`...`	Reserved for future use.

Value

A data frame with one row per column chunk, ordered by row group and then by column, with the columns:

`row_group` 1-based row-group ID, as `row_groups()` reports it.

`column` 1-based physical column index, as `schema()` reports it.

`path` Complete dotted column path; see `schema()` for why this is the identifier rather than the bare leaf name.

`physical_type` Parquet physical type of the leaf.

`compression` Codec name, such as `"SNAPPY"` or `"UNCOMPRESSED"`. Set per chunk, so one file may mix codecs.

`num_values` Values stored in the chunk, nulls included.

`compressed_bytes`, `uncompressed_bytes` Size of the chunk on disk and after decompression. Equal for an uncompressed chunk.

`encodings` Encodings the chunk declares, comma separated. A dictionary-encoded chunk typically lists `RLE_DICTIONARY` alongside the `PLAIN` its dictionary page uses.

`dictionary_page` Whether the chunk has a dictionary page.

`bloom_filter` Whether a bloom filter is present, which is what `bloom_filter_may_contain()` needs.

`page_index` Whether a column index or an offset index is present; `page_index()` reports either.

Sizes and counts are doubles rather than integers, because a chunk can exceed `.Machine$integer.max`.

See Also

`column_statistics()`, `row_groups()`

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(mtcars, path, row_group_size = 16)
pf <- open_parquet(path)
column_chunks(pf)
close_parquet(pf)
```

`column_statistics` *Inspect Parquet column statistics*

Description

Reports the per-column, per-row-group statistics recorded in the file: value and null counts, and the minimum and maximum bounds.

Usage

```
column_statistics(x, ...)

## S3 method for class 'qio_parquet_file'
column_statistics(x, ...)
```

Arguments

`x`	A `qio_parquet_file` object.
`...`	Reserved for future use.

Details

These are **claims made by whoever wrote the file**, not facts qio verifies. A reader that skips a row group on them is trusting that writer. qio does not use them to skip anything.

`min` and `max` are list columns, because one file can hold columns of different types. Each element holds the bound decoded at the *physical* level: an `INT64` bound stays a number rather than becoming a `POSIXct`, and a decimal is not scaled, since a bound is a sort key rather than a value to compute with. Text columns are the exception and decode to character. An element is `NULL` when the bound is absent, or is present but the wrong width for its type.

Value

A data frame with one row per column chunk, ordered by row group and then by column, with the columns:

`row_group` 1-based row-group ID, as `row_groups()` reports it.

`column` 1-based physical column index, as `schema()` reports it.

`path` Complete dotted column path; see `schema()`.

`num_values` Values the statistics cover, nulls included.

`null_count` Nulls the writer recorded, or `NA` when it recorded none. `NA` means unknown, not zero.

`distinct_count` Distinct values the writer recorded, or `NA`. Most writers omit it, so `NA` is the common case.

`min`, `max` List columns of physical-level bounds, one element per row; `NULL` when absent or undecodable. See Details.

Counts are doubles rather than integers, because a chunk can hold more values than `.Machine$integer.max`.

See Also

`column_chunks()`, `row_groups()`

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(data.frame(n = 1:100), path, row_group_size = 25)
pf <- open_parquet(path)
stats <- column_statistics(pf)
stats[c("row_group", "path", "null_count")]
unlist(stats$min)
close_parquet(pf)
```

infer_parquet_schema *Infer the Parquet writer schema for an R object*

Description

Shows the physical and logical Parquet types that `write_parquet()` would use without an explicit schema. Nullability is inferred from missing values.

Usage

```
infer_parquet_schema(x)
```

Arguments

`x`	A data frame or list of equal-length atomic vectors.

Value

A `qio_parquet_schema` data frame with one row per column.

Examples

```
infer_parquet_schema(data.frame(id = 1:3, when = as.Date("2020-01-01")))
```

metadata *Inspect Parquet footer metadata*

Description

Duplicate keys are preserved in their original order.

Usage

```
metadata(x, ...)

## S3 method for class 'qio_parquet_file'
metadata(x, ...)
```

Arguments

`x`	A `qio_parquet_file` object.
`...`	Reserved for future use.

Value

A data frame with `key` and `value` columns.

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(mtcars, path)
pf <- open_parquet(path)
metadata(pf)
close_parquet(pf)
```

`open_parquet` *Open a Parquet file*

Description

Opens a Parquet file for inexpensive metadata inspection and selective, batched reading. The returned handle is valid only in the current R session.

Usage

```
open_parquet(file, mmap = FALSE, verify_checksums = TRUE, threads = NULL)
```

Arguments

`file`	Path to a Parquet file, or an `http://`, `https://`, `ftp://`, `ftps://` or `file://` URL. A URL is downloaded to the session temporary directory in full before any of it is read; the copy is removed by `close_parquet()`, so a handle opened from a URL must be closed to reclaim the space. See qio-limitations.
`mmap`	Use memory-mapped input. On Windows a path the active code page cannot represent is read with buffered input instead, because only the mapped path needs a name that page can express. The result is the same.
`verify_checksums`	Verify Parquet page checksums when present.
`threads`	Number of reader threads. `NULL` (the default) and `0` both use two threads. Larger explicit values are honored in ordinary use and capped at two when R requests CRAN-compatible core limits. `collect()` decodes columns in parallel either way: a mapped file shares one reader, and a buffered one gives each worker its own. Pass `threads = 1` to force serial reads.

Value

A `qio_parquet_file` object. Close it with `close_parquet()`.

See Also

`collect()` and `walk_batches()` to read from the handle, `close_parquet()` to release it, `schema()` and `metadata()` to inspect it without reading, and `read_parquet()` for a whole file in one call.

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(mtcars, path)
pf <- open_parquet(path)
dim(pf)
close_parquet(pf)
```

`page_index` *Inspect Parquet page indexes*

Description

Reports the per-page index of each column chunk: where each data page sits in the file, which row it starts at, and the bounds and null count it declares. One row per page.

Usage

```
page_index(x, ...)

## S3 method for class 'qio_parquet_file'
page_index(x, ...)
```

Arguments

`x`	A `qio_parquet_file` object.
`...`	Reserved for future use.

Details

A page index is optional, and many writers omit it. Columns without one contribute no rows, so a file with no page index at all returns a zero-row data frame rather than an error. qio's own writer does not emit page indexes; see qio-limitations.

Like `column_statistics()`, the bounds are claims made by whoever wrote the file. qio reports them and does not use them to skip pages.

Value

A data frame with one row per page, ordered by row group, then column, then page, with the columns:

`row_group` 1-based row-group ID, as `row_groups()` reports it.

`column` 1-based physical column index, as `schema()` reports it.

`path` Complete dotted column path; see `schema()`.

`page` 1-based page number within this column chunk, restarting at 1 for every chunk.

`first_row` 0-based row within the row group where the page starts. The first page of a chunk is 0.

`offset` Byte offset of the page from the start of the file.

`compressed_bytes` Size of the page on disk.

`null_count` Nulls on the page, or `NA` when not recorded.

`null_page` Whether the page holds only nulls, in which case its bounds carry no information.

`min`, `max` List columns of physical-level bounds, decoded as in `column_statistics()`; `NULL` when absent.

The two indexes are independent and either may be missing. `first_row`, `offset`, and `compressed_bytes` come from the offset index and are `NA` without it; `null_count`, `null_page`, `min`, and `max` come from the column index and are `NA` or `NULL` without it.

See Also

`column_statistics()`, `column_chunks()`

Examples

```
# qio does not write page indexes, so a file it wrote has none and the
# result is empty rather than an error.
path <- tempfile(fileext = ".parquet")
write_parquet(data.frame(n = 1:10), path)
pf <- open_parquet(path)
nrow(page_index(pf))
close_parquet(pf)
```

---

parquet_schema *Create a Parquet writer schema*

Description

Creates a reusable schema that controls how `write_parquet()` stores selected columns. Each argument must be named and may be a type string or a list whose first element is the type. Schemas may be partial: unspecified columns keep qio's automatic mapping.

Usage

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

Arguments

`...`	Named Parquet type specifications.

Details

Supported declarations are `"AUTO"`, `"BOOLEAN"`, `"INT32"`, `"INT64"`, `"FLOAT"`, `"DOUBLE"`, `"STRING"`, `"DATE"`, and `"TIMESTAMP"`. `TIMESTAMP` accepts `unit` (`"MILLIS"`, `"MICROS"`, or `"NANOS"`) and must be adjusted to UTC. All types accept `repetition_type` (`"AUTO"`, `"REQUIRED"`, or `"OPTIONAL"`).

`INT32` and `INT64` declarations accept finite whole-number inputs; `INT64` is limited to R's exact double-integer range from `-2^53` through `2^53`. `FLOAT` and `DOUBLE` accept numeric input, `STRING` accepts character or factor input, `DATE` accepts `Date` or whole-number days, and `TIMESTAMP` accepts `POSIXct`.

Value

A qio_parquet_schema data frame.

Examples

```
parquet_schema(
  id = "INT64",
  price = "FLOAT",
  created_at = list("TIMESTAMP", unit = "MILLIS")
)
```

`parquet_type_mapping` *Show Parquet physical type mappings*

Description

Lists every Parquet physical type and the R storage type qio uses for it **when the column carries no logical annotation**. Missing values indicate unsupported mappings.

Usage

```
parquet_type_mapping()
```

Details

These are physical fallbacks, and most real columns are annotated, so this table is not a prediction of what a given file will produce. Use `read_plan()` for that: it resolves the annotation, the `int64`, `time`, and `tz` options, and reports the R type each column will actually materialize as.

Two entries are worth reading carefully. `BYTE_ARRAY` and `FIXED_LEN_BYTE_ARRAY` are bytes here, returned as a list of raw vectors, because bytes are only text when the file says so; a `STRING`, `ENUM`, or `JSON` annotation is what makes a `BYTE_ARRAY` character. `INT96` is a deprecated physical type used only for timestamps, so it is read as `POSIXct` with no annotation involved.

Value

A data frame with one row per Parquet physical type and the columns:

`physical_type` Parquet physical type, spelled as `schema()` and `read_plan()` report it.

`r_type` R type produced with no logical annotation, spelled as `read_plan()` reports it. `NA` when the type cannot be read.

`written_from` R input that infers this physical type, or `NA` when `write_parquet()` cannot produce it.

See Also

`read_parquet()`, `write_parquet()`, `schema()`

Examples

```
parquet_type_mapping()
```

`qio-limitations` *What qio does not do*

Description

The vendored `carquet` library implements more of the Parquet specification than qio exposes. This topic records what is deliberately left out and why, so an absent function reads as a decision rather than an oversight.

Not exposed in 0.1.0

Nested and repeated columns `LIST`, `MAP`, and struct columns are skipped on read, with one message per operation. carquet returns the definition and repetition levels needed to rebuild them, but assembling R objects from those levels, and deciding how a null list differs from an empty one, is qio's work and is deferred to 0.2.0.

Predicate pushdown and page filters carquet can skip pages using statistics. qio reads whole columns. Filtering happens in R, after the read, where the answer does not depend on a writer's honesty about its own statistics.

Writing bloom filters and page indexes Both can be *read*: `bloom_filter_may_contain()` tests membership and `page_index()` reports per-page bounds and locations. Neither is written, because the writer options that control them belong to the writer configuration deferred below.

Reading back a declared sort order `write_parquet()` can record one with `sorted_by`, but carquet exposes no way to read it back, so qio cannot report the declaration in a file it did not write.

Encryption Files with an encrypted footer are rejected by `validate_parquet()` and cannot be read.

External column metadata Modelled by carquet but not implemented there; the API returns "not implemented".

Writer tuning Dictionary encoding, per-column encodings, page sizes, checksums, and index generation are carquet options that qio does not surface. The reusable writer configuration that would carry them is deferred to 0.2.0 rather than guessed at now.

Geospatial and variant types Read as their physical storage, without interpretation.

Partial reads over HTTP A URL is supported by downloading the whole file to the session temporary directory first, so selecting columns or row groups saves decoding but not transfer. Reading only the footer and the chosen column chunks needs HTTP range requests, and carquet accepts input only as a path, a `FILE*`, or a buffer – there is no way to supply read and seek callbacks, so there is no seam for range requests to reach it. That needs a custom IO interface in carquet itself and is deferred to 0.2.0.

Boundaries that are not carquet's

Some limits come from R rather than from Parquet. A single result cannot exceed `.Machine$integer.max` rows. `INT64` columns lose precision beyond 2^53 unless read as bit64::integer64. R's `integer` reserves `-2147483648` for `NA`, so a Parquet `INT32` holding that value reads as `NA` with a warning. See qio-types and `collect()` for the details.

See Also

`column_chunks()`, `validate_parquet()`, `collect()`

`qio-types` *Parquet type mapping*

Description

The complete mapping between Parquet types and R, in one table: what each physical type carries, which logical annotations qio applies to it, what reading produces, and whether writing can produce it.

Details

A Parquet column has a *physical* type, which is how its bytes are stored, and optionally a *logical* annotation, which says what those bytes mean. The annotation decides the R type wherever qio implements one; the physical type is the fallback when there is no annotation or qio does not implement it.

`read_plan()` answers the same question for one real file, including the effect of the `int64`, `time`, and `tz` arguments. Prefer it when you have the file in hand; this table is the general contract.

Read and write mapping

Reading covers the whole table. Writing is deliberately narrower: qio writes the types R can express unambiguously, and an explicit `parquet_schema()` selects among them. Everything marked "no" under writing is readable but not writable.

Physical	Logical	Reads as	Notes and precision
`BOOLEAN`	none	`logical`	Exact.
`INT32`	none	`integer`	`-2147483648` becomes `NA`: R reserves it
`INT32`	`DATE`	`Date`	Exact. Days since 1970-01-01.
`INT32`	`TIME(MILLIS)`	`double` or `hms`	Seconds since midnight. Never `POSIXct`
`INT32`	`INTEGER(8/16/32, signed)`	`integer`	Exact. The sentinel rule above applies.
`INT32`	`INTEGER(8/16, unsigned)`	`integer`	Exact; both fit in R's signed 32-bit integ
`INT32`	`INTEGER(32, unsigned)`	`double`	Exact. Widened so the upper half stays p
`INT32`	`DECIMAL(p, s)`	`double`	Scale applied, so unscaled `1230` scale 2
`INT64`	none	`double`, or `integer64`	Default `int64 = "double"` is exact in [-
`INT64`	`TIMESTAMP(unit, UTC)`	`POSIXct`	An instant; `tz` changes only display. Sto
`INT64`	`TIMESTAMP(unit, local)`	`POSIXct`	A wall clock with no zone stored. Civil

INT64	TIME(MICROS/NANOS)	double or hms	As INT32 TIME above.
INT64	INTEGER(64, signed)	double or integer64	As bare INT64.
INT64	INTEGER(64, unsigned)	double or integer64	Never negative. Exact to 2^53 in double
INT64	DECIMAL(p, s)	double	As INT32 DECIMAL.
INT96	none	POSIXct	Deprecated; only ever a timestamp. Rea
FLOAT	none	double	Exact: every 32-bit float is representable
DOUBLE	none	double	Exact.
BYTE_ARRAY	STRING, ENUM, JSON	character	Validated as UTF-8; invalid bytes fail wi
BYTE_ARRAY	DECIMAL(p, s)	double	Big-endian two's complement, scale app
BYTE_ARRAY	none, BSON, other	list of raw	Arbitrary bytes stay bytes; NULL for null
FIXED_LEN_BYTE_ARRAY	UUID	character	Canonical hyphenated form from exactly
FIXED_LEN_BYTE_ARRAY	FLOAT16	double	Exact: every half-precision value is repr
FIXED_LEN_BYTE_ARRAY	DECIMAL(p, s)	double	As BYTE_ARRAY DECIMAL.
FIXED_LEN_BYTE_ARRAY	none, INTERVAL, other	list of raw	Fixed-width raw vectors, each validated
any	NULL	logical	All NA, whatever the physical type: the a
any	LIST, MAP, struct	not read	Skipped with one message per operation

Where precision is lost

Four cases lose information, all of them because R's types are narrower than Parquet's. Each is reported rather than silent.

INT32 **holding** -2147483648 R reserves that value as NA_integer_, so it cannot be stored. It reads as NA with one warning naming the column, however many values or batches were affected. Promoting the column to double was rejected: read_plan() is a pure function of the schema, and a data-dependent type would let one column arrive as different types in different batches.

64-bit integers past 2^53 R's double is exact only within [-2^53, 2^53]. Values outside it read as NA with one warning naming the column; pass int64 = "integer64" to keep the signed range, except -9223372036854775808, which bit64 reserves as its own NA and which therefore reads as NA in either mode.

DECIMAL Read as double with the declared scale applied, which is exact only while the unscaled integer stays within [-2^53, 2^53]. Larger precisions lose low-order digits. One message per read. Exact fixed-point reads are planned for 0.2.0.

Sub-second timestamps far from the epoch POSIXct is a double of seconds, so the further a timestamp is from 1970 the less sub-second precision survives. A NANOS column shows this first.

Writing loses nothing that reading would not: qio writes only the types it can represent exactly. POSIXct is the one rounding case, to the declared timestamp unit, which defaults to microseconds.

What writing supports

Without a schema, qio infers: logical to BOOLEAN, integer to INT32, double to DOUBLE, character and factor to BYTE_ARRAY with STRING, Date to INT32 with DATE, and POSIXct to INT64 with a UTC-adjusted TIMESTAMP in microseconds.

An explicit parquet_schema() may additionally select INT64 or FLOAT for a numeric column, and a different TIMESTAMP unit. Those are the only combinations the writer accepts; anything else is an error rather than a silent fallback.

A column is written OPTIONAL when it contains any NA and REQUIRED otherwise, except when appending, where the existing file decides. For double columns NA becomes a Parquet null while NaN is preserved as a value.

See Also

read_plan() for one file, parquet_type_mapping() for the physical fallbacks alone, qio-limitations for what is out of scope.

---

read_parquet *Read a Parquet file*

Description

Reads an Apache Parquet file into a data frame.

Usage

```
read_parquet(
  file,
  ...,
  columns = NULL,
  row_groups = NULL,
  int64 = c("double", "integer64"),
  time = c("numeric", "hms"),
  tz = "UTC",
  verbose = FALSE
)
```

Arguments

file	Path to a Parquet file, or an http://, https://, ftp://, ftps:// or file:// URL. A URL is downloaded to the session temporary directory in full before any of it is read, and the copy is removed when the read finishes; see qio-limitations.
...	Must be empty. Every argument after it is name-only, matching collect(), walk_batches() and read_plan(), which take the same arguments the same way.
columns	Character vector of complete column paths, or NULL (the default) for all columns; see collect().
row_groups	Integer vector of 1-based row-group IDs, or NULL (the default) for all row groups.
int64	How 64-bit integer columns reach R; see collect().
time	How TIME columns reach R; see collect().
tz	Time zone for TIMESTAMP columns; see collect().
verbose	Report the read plan before reading; see collect().

Details

Column types are mapped from Parquet as follows: `BOOLEAN` to logical, `INT32` to integer, and `INT64`/`FLOAT`/`DOUBLE` to double. A `BYTE_ARRAY` becomes character only when the file annotates it `STRING`, `ENUM`, or `JSON`; without an annotation it is arbitrary bytes and is returned as a list of raw vectors, because assuming UTF-8 the file never claimed would corrupt binary data. An `INT32` column annotated `DATE` is returned as a `Date`, a UTC-adjusted `TIMESTAMP` (physical `INT64`) as a `POSIXct` in UTC, and a legacy `INT96` timestamp as a `POSIXct` in UTC (interpreting its Julian-day and nanosecond-of-day parts as an instant). Parquet nulls become `NA`. `INT64` values are returned as doubles and lose precision beyond 2^53, which for microsecond and nanosecond timestamps can drop sub-second precision far from the epoch. Use `read_plan()` to preview the R type of each column before reading.

Nested and repeated columns are skipped with one message. Nested reading is deferred to qio 0.2.0.

The file is memory-mapped for the duration of the read (falling back to buffered reads if mapping fails) so columns decode in parallel; the mapping is released before the function returns.

`columns` and `row_groups` read part of a file and are passed straight to `collect()`. Selecting columns is the single largest speedup available on a wide file, because a column that is not selected is never decompressed. Use `open_parquet()` with `collect()` for the rest: `batch_size`, `mmap`, `threads`, and `verify_checksums`.

Value

A data frame.

See Also

`write_parquet()`, `open_parquet()` and `collect()` to read part of a file, `walk_batches()` for a file larger than memory, and `read_plan()` to preview the R type of every column before reading.

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(data.frame(x = 1:3, y = c("a", "b", NA)), path)
read_parquet(path)
```

`read_plan` *Plan how a Parquet file is read into R*

Description

Builds a read plan from a Parquet schema: one row per physical leaf column describing the R type each column will materialize as, whether it can be collected, and why not when it cannot. The plan is a pure function of the schema, so it is cheap to compute and inspect before reading any data.

Usage

```
read_plan(x, ...)

## S3 method for class 'qio_parquet_file'
read_plan(
  x,
  ...,
  int64 = c("double", "integer64"),
  time = c("numeric", "hms"),
  tz = "UTC"
)

## S3 method for class 'character'
read_plan(
  x,
  ...,
  int64 = c("double", "integer64"),
  time = c("numeric", "hms"),
  tz = "UTC"
)

## S3 method for class 'data.frame'
read_plan(
  x,
  ...,
  int64 = c("double", "integer64"),
  time = c("numeric", "hms"),
  tz = "UTC"
)
```

Arguments

x	A Parquet file path, a qio_parquet_file object, or the data frame returned by schema().
...	Reserved for future use.
int64	How 64-bit integer columns reach R; see collect(). The plan reports the resulting r_type and converter, so it can be inspected for exactly the read that will follow.
time	How TIME columns reach R; see collect().
tz	Time zone for TIMESTAMP columns; see collect().

Details

The plan reflects what collect(), read_parquet(), and walk_batches() actually do today. A logical annotation overrides the physical fallback in parquet_type_mapping(), and the annotations qio resolves are:

- DATE to Date, and legacy physical INT96 to POSIXct.

- TIMESTAMP to POSIXct, UTC-adjusted or interpreted in tz.
- TIME to seconds or hms::hms, selected by time.
- INTEGER at any width and sign, including unsigned 64-bit, which with INT64 is selected by int64.
- STRING, ENUM, and JSON to character; everything else stored as bytes stays a list of raw vectors.
- UUID to canonical text and FLOAT16 to double.
- DECIMAL to double with the scale applied, from either integer or binary storage.

Unimplemented annotations are reported in note and retain their physical fallback type. converter names the exact conversion the reader will run, so it distinguishes cases that share an r_type.

A path is enough – read_plan() opens the file, reads the footer, and closes it again, so no handle is needed to inspect a file before reading it. Passing an open open_parquet() handle, or the data frame from schema(), produces exactly the same plan; use those when a handle is already open or when the schema has already been fetched.

Pass the same int64, time, and tz the read will use. The plan resolves them, so r_type and converter describe that read rather than a default one.

Value

A qio_read_plan data frame with one row per physical leaf column and the columns:

column 1-based physical column index.

name, path Column name and dotted path.

physical_type, logical_type Parquet physical type and logical annotation (NA when absent).

r_type Target R type, or NA when the column cannot be collected.

converter Stable identifier of the conversion the reader uses.

nullable Whether the column can contain nulls.

nested Whether the leaf belongs to a nested or repeated field.

collectible Whether collect() can currently materialize the column.

note Reason a column is not collectible, or a pending logical annotation; NA otherwise.

See Also

schema(), collect(), parquet_type_mapping()

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(data.frame(x = 1:3, y = c("a", "b", NA)), path)

# A path is enough; no handle is needed.
read_plan(path)

# The plan answers for the read you are about to do, not a default one.
read_plan(path, int64 = "integer64")
```

```
# An open handle and a schema() data frame give the same plan.
pf <- open_parquet(path)
identical(read_plan(pf), read_plan(path))
close_parquet(pf)
```

`row_groups` *Inspect Parquet row groups*

Description

Inspect Parquet row groups

Usage

```
row_groups(x, ...)

## S3 method for class 'qio_parquet_file'
row_groups(x, ...)
```

Arguments

`x`	A `qio_parquet_file` object.
`...`	Reserved for future use.

Value

A data frame with one row per row group and the columns:

`row_group` 1-based row-group ID, which is what `row_groups =` selects on in `collect()`, `read_parquet()`, and `walk_batches()`.

`rows` Rows in the group.

`compressed_bytes`, `uncompressed_bytes` Total size of the group's column chunks on disk and after decompression.

Counts and sizes are doubles rather than integers, because a row group can exceed `.Machine$integer.max`.

See Also

`column_chunks()` for the same sizes per column, and `column_statistics()` for what the writer claims about each chunk.

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(mtcars, path)
pf <- open_parquet(path)
row_groups(pf)
close_parquet(pf)
```

schema *Inspect a Parquet schema*

Description

Reports every physical leaf column in the file, in file order.

Usage

```
schema(x, ...)

## S3 method for class 'qio_parquet_file'
schema(x, ...)
```

Arguments

x	A qio_parquet_file object.
...	Reserved for future use.

Details

name and path are not interchangeable. name is the bare leaf name and is not unique: two leaves under different parents may share one, and a map's key/value leaves routinely do. path is the complete dotted path and is what identifies a column everywhere else in qio – collect(), read_parquet(), walk_batches(), and bloom_filter_may_contain() all select by path, and column_chunks(), column_statistics(), and page_index() report it under the same name.

Value

A data frame with one row per physical leaf column and the columns:

column 1-based physical column index.

name Bare leaf name; not unique. See Details.

path Complete dotted path; unique, and what selection uses.

physical_type Parquet physical type.

logical_type Logical annotation, or NA when absent.

logical_details Annotation parameters, such as a timestamp unit or a decimal precision and scale; NA when there are none.

repetition_type "REQUIRED", "OPTIONAL", or "REPEATED".

type_length Declared width of a FIXED_LEN_BYTE_ARRAY, else 0.

max_definition_level Above 0 when the leaf is nullable.

max_repetition_level Above 0 when the leaf is repeated.

See Also

`read_plan()` for the R type each column will produce, `column_chunks()` for how each is stored, and `parquet_type_mapping()` for the physical fallbacks.

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(mtcars, path)
pf <- open_parquet(path)
schema(pf)
close_parquet(pf)
```

---

validate_parquet *Check that a file is structurally valid Parquet*

Description

Reports why a file cannot be read, in terms of the file rather than of the parser. Opening a damaged or misidentified file otherwise fails somewhere inside footer parsing, with a message that describes a byte offset instead of the problem.

Usage

```
validate_parquet(file)
```

Arguments

`file`	Path to a file, or a URL as in `read_parquet()`. A downloaded copy is removed before this returns.

Details

What is checked: the file exists and is large enough to be Parquet, both magic markers are present, the footer parses, and the schema and row-group metadata agree with the file's own row count.

What is **not** checked: the data pages. Structural validity says a reader can find the columns, not that their bytes decode or that the recorded statistics are true. To check the pages, read the file with `open_parquet(verify_checksums = TRUE)` and `collect()`; that costs a full read, which is why it is not done here.

Value

`TRUE`, invisibly. Raises an error describing the first problem found otherwise.

See Also

`open_parquet()`, `column_chunks()`

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(mtcars, path)
validate_parquet(path)

# A file that is not Parquet at all.
plain <- tempfile()
writeLines("not parquet", plain)
try(validate_parquet(plain))
```

`walk_batches` *Walk over batches from a Parquet file*

Description

Calls `FUN(batch, index, ...)` for every batch. Each batch is an independent data frame and can be retained by the callback when desired. Callback return values are discarded.

Usage

```
walk_batches(
  x,
  FUN,
  ...,
  columns = NULL,
  row_groups = NULL,
  batch_size = 65536L,
  int64 = c("double", "integer64"),
  time = c("numeric", "hms"),
  tz = "UTC",
  verbose = FALSE
)
```

Arguments

`x`	A `qio_parquet_file` object.
`FUN`	Function called with a data frame and a 1-based global batch index, followed by `...`.
`...`	Passed on to `FUN` after the batch and its index. This differs from `collect()`, where `...` must be empty: here it is how a callback receives extra arguments. Every argument after it is still name-only.
`columns`	Character vector of complete column paths, or `NULL` for all columns. Paths are matched exactly as `schema()` and `names()` report them, dot-separated for nested leaves, and never by leaf name alone: two leaves may share a name under different parents. An unknown path is an error, and so is a path matching more than one leaf.

row_groups — Integer vector of 1-based row-group IDs, or NULL for all row groups.

batch_size — Positive number of rows decoded per batch.

int64 — How 64-bit integer columns reach R. "double" (the default) is exact from -2^53 through 2^53 and returns NA outside it. "integer64" returns bit64::integer64, which needs the suggested bit64 package and covers the signed 64-bit range **except its lowest value**: bit64 reserves -9223372036854775808 as its own NA, so a column storing INT64_MIN reads as NA in either mode. Either way values that cannot be represented become NA, and one warning naming the column is emitted for each column that lost values – once per column, however many values, row groups, or batches were affected, and never for a column that lost nothing. Unsigned 64-bit columns are never returned as negative numbers.

time — How TIME columns reach R: "numeric" (the default) returns seconds since midnight, "hms" returns hms::hms and needs the suggested hms package. Neither returns POSIXct, because a time of day is not an instant.

tz — Time zone name, "UTC" by default. A UTC-adjusted TIMESTAMP is an instant, so tz changes only how it prints. A non-UTC TIMESTAMP is a wall clock with no zone stored, so its civil components are interpreted in tz; base R decides ambiguous and nonexistent times at daylight-saving boundaries. The machine's local zone is never used implicitly.

verbose — Report what the read is about to do before doing it: the rows, columns, and row groups selected, the batch size, and the resolved read_plan() for the selected columns only – not for the whole file, so it answers "what am I about to get". Written with message(), so it goes to stderr and suppressMessages() silences it.

Value

x, invisibly.

See Also

collect() for the same selection returned as one data frame, and open_parquet() for the handle.

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(mtcars, path)
pf <- open_parquet(path)
walk_batches(pf, function(batch, index) print(head(batch)))
close_parquet(pf)
```

`write_parquet` *Write a Parquet file*

Description

Writes a data frame to an Apache Parquet file.

Usage

```
write_parquet(
  x,
  file,
  compression = c("snappy", "zstd", "gzip", "lz4", "uncompressed"),
  schema = NULL,
  row_group_size = NULL,
  metadata = NULL,
  sorted_by = NULL,
  append = FALSE
)
```

Arguments

`x`	A data frame (or a list of equal-length atomic vectors).
`file`	Output path. Must be local: qio reads from a URL but cannot write to one.
`compression`	Compression codec: one of `"snappy"` (default), `"zstd"`, `"gzip"`, `"lz4"`, or `"uncompressed"`.
`schema`	An optional schema created by `parquet_schema()`. Named entries override qio's inferred mapping; omitted columns retain automatic mapping.
`row_group_size`	Rows per row group, or `NULL` (default) to write a single row group. Smaller groups let other readers skip more but add per-group metadata and can compress worse.
`metadata`	A named character vector of footer key/value metadata, or `NULL`. Duplicate keys are written in the order given. An `NA` value is written as a key with no value, and reads back as `NA`.
`sorted_by`	Columns the data is already sorted by, or `NULL`. Either a character vector of column names, meaning ascending with nulls last, or a data frame with a `name` column and optional logical `descending` and `nulls_first` columns. This **records a claim and nothing more**: qio does not sort the data and does not check that the claim is true. A wrong declaration misleads every reader that trusts it.
`append`	Append new row groups to an existing file instead of replacing it. The file must already exist and describe exactly the columns being written; see Details.

Details

Supported column types are logical, integer, double, character, and factor (written as character). A column is written as nullable when it contains any NA. Date columns are written as INT32 with a DATE annotation, and POSIXct columns as INT64 microseconds with a UTC-adjusted TIMESTAMP annotation; both round-trip back to their R class. Sub-microsecond fractions of a second are rounded. Other classed columns are still written using their underlying storage type and lose their class. An explicit parquet_schema() may instead select INT64, FLOAT, or a different timestamp unit, among the supported declarations.

append adds row groups to a file that already exists, rather than replacing it. Because it writes into data the user already has, qio checks compatibility itself and refuses anything it cannot prove safe. The bundled library compares column count, order, names, physical types, repetition type, and logical type *identity* – but not logical *parameters*, so it would happily append microsecond timestamps to a millisecond file, or a decimal of one scale to another. qio compares the full declaration, including those parameters, and the schema path of every column.

Nullability is taken from the existing file rather than inferred from the new data, so appending a batch that happens to contain no NA to a nullable column works. Appending data that does contain NA to a column the file declares REQUIRED is refused.

Row groups are the unit other readers skip on: a reader that can rule a group out from its statistics never touches its pages. row_group_size sets how many rows go in each. The default writes one row group, which keeps files compact but leaves nothing to skip, so a file meant to be filtered by other tools should set it.

metadata writes application key/value pairs into the footer, where metadata() reads them back. Keys and values are stored as UTF-8 text; Parquet defines no meaning for them.

Value

The output path, invisibly.

See Also

read_parquet(), metadata(), row_groups()

Examples

```
path <- tempfile(fileext = ".parquet")
write_parquet(mtcars, path)

# Several row groups, with provenance in the footer.
write_parquet(
  mtcars,
  path,
  row_group_size = 8,
  metadata = c(source = "mtcars", written_by = "qio")
)
```

Index

bit64::integer64, *5*, *15*, *25*
bloom_filter_may_contain, 2
bloom_filter_may_contain(), *7*, *14*, *22*

close_parquet, 3
close_parquet(), *10*
collect, 4
collect(), *10*, *15*, *17–25*
column_chunks, 6
column_chunks(), *3*, *8*, *12*, *15*, *21–23*
column_statistics, 7
column_statistics(), *7*, *11*, *12*, *21*, *22*

hms::hms, *5*, *20*, *25*

infer_parquet_schema, 9

metadata, 9
metadata(), *10*, *27*

names(), *2*, *5*, *24*

open_parquet, 10
open_parquet(), *4*, *6*, *18*, *20*, *23*, *25*

page_index, 11
page_index(), *7*, *14*, *22*
parquet_schema, 12
parquet_schema(), *15*, *16*, *26*, *27*
parquet_type_mapping, 13
parquet_type_mapping(), *17*, *19*, *20*, *23*

qio-limitations, *3*, *10*, *11*, 14, *17*
qio-types, 15, *15*

read_parquet, 17
read_parquet(), *6*, *10*, *13*, *19*, *21–23*, *27*
read_plan, 18
read_plan(), *5*, *6*, *13*, *15–18*, *23*, *25*
row_groups, 21
row_groups(), *6–8*, *11*, *27*

schema, 22
schema(), *2*, *5*, *6*, *8*, *10*, *11*, *13*, *19*, *20*, *24*

validate_parquet, 23
validate_parquet(), *14*, *15*

walk_batches, 24
walk_batches(), *5*, *6*, *10*, *17–19*, *21*, *22*
write_parquet, 26
write_parquet(), *9*, *12–14*, *18*