Package 'fst'

February 8, 2022

Type Package

Title Lightning Fast Serialization of Data Frames

Description Multithreaded serialization of compressed data frames using the 'fst' format. The 'fst' format allows for full random access of stored data and a wide range of compression settings using the LZ4 and ZSTD compressors.

Version 0.9.8

Date 2022-02-07

Depends R (>= 3.0.0)

Imports fstcore, Rcpp

LinkingTo Rcpp, fstcore

SystemRequirements little-endian platform

RoxygenNote 7.1.2

Suggests testthat, bit64, data.table, lintr, nanotime, crayon

License AGPL-3 | file LICENSE

Encoding UTF-8

URL http://www.fstpackage.org

BugReports https://github.com/fstpackage/fst/issues

NeedsCompilation yes

Author Mark Klik [aut, cre, cph]

Maintainer Mark Klik <markklik@gmail.com>

Repository CRAN

Date/Publication 2022-02-08 07:40:02 UTC

R topics documented:

`fst-package` *Lightning Fast Serialization of Data Frames for R.*

Description

Multithreaded serialization of compressed data frames using the 'fst' format. The 'fst' format allows for random access of stored data which can be compressed with the LZ4 and ZSTD compressors.

Details

The fst package is based on three C++ libraries:

- **fstlib**: library containing code to write, read and compute on files stored in the *fst* format. Written and owned by Mark Klik.
- **LZ4**: library containing code to compress data with the LZ4 compressor. Written and owned by Yann Collet.
- **ZSTD**: library containing code to compress data with the ZSTD compressor. Written by Yann Collet and owned by Facebook, Inc.

As of version 0.9.8, these libraries are included in the fstcore package, on which fst depends. The copyright notices of the above libraries can be found in the fstcore package.

`compress_fst` *Compress a raw vector using the LZ4 or ZSTD compressor.*

Description

Compress a raw vector using the LZ4 or ZSTD compressor.

Usage

```
compress_fst(x, compressor = "ZSTD", compression = 0, hash = FALSE)
```

Arguments

x	raw vector.
compressor	compressor to use for compressing x. Valid options are "LZ4" and "ZSTD" (default).
compression	compression factor used. Must be in the range 0 (lowest compression) to 100 (maximum compression).
hash	Compute hash of compressed data. This hash is stored in the resulting raw vector and can be used during decompression to check the validity of the compressed vector. Hash computation is done with the very fast xxHash algorithm and implemented as a parallel operation, so the performance hit will be very small.

---

decompress_fst *Decompress a raw vector with compressed data.*

Description

Decompress a raw vector with compressed data.

Usage

```
decompress_fst(x)
```

Arguments

x	raw vector with data previously compressed with compress_fst.

Value

a raw vector with previously compressed data.

---

fst *Access a fst file like a regular data frame*

Description

Create a fst_table object that can be accessed like a regular data frame. This object is just a reference to the actual data and requires only a small amount of memory. When data is accessed, only a subset is read from file, depending on the minimum and maximum requested row number. This is possible because the fst file format allows full random access (in columns and rows) to the stored dataset.

Usage

```
fst(path, old_format = FALSE)
```

Arguments

path	path to fst file
old_format	must be FALSE, the old fst file format is deprecated and can only be read and converted with fst package versions 0.8.0 to 0.8.10.

Value

An object of class fst_table

Examples

```
## Not run:
# generate a sample fst file
path <- paste0(tempfile(), ".fst")
write_fst(iris, path)

# create a fst_table object that can be used as a data frame
ft <- fst(path)

# print head and tail
print(ft)

# select columns and rows
x <- ft[10:14, c("Petal.Width", "Species")]

# use the common list interface
ft[TRUE]
ft[c(TRUE, FALSE)]
ft[["Sepal.Length"]]
ft$Petal.Length

# use data frame generics
nrow(ft)
ncol(ft)
dim(ft)
dimnames(ft)
colnames(ft)
rownames(ft)
names(ft)

## End(Not run)
```

---

hash_fst *Parallel calculation of the hash of a raw vector*

Description

Parallel calculation of the hash of a raw vector

Usage

```
hash_fst(x, seed = NULL, block_hash = TRUE)
```

Arguments

x	raw vector that you want to hash
seed	The seed value for the hashing algorithm. If NULL, a default seed will be used.
block_hash	If TRUE, a multi-threaded implementation of the 64-bit xxHash algorithm will be used. Note that block_hash = TRUE or block_hash = FALSE will result in different hash values.

Value

hash value

---

metadata_fst *Read metadata from a fst file*

Description

Method for checking basic properties of the dataset stored in path.

Usage

```
metadata_fst(path, old_format = FALSE)

fst.metadata(path, old_format = FALSE)
```

Arguments

path	path to fst file
old_format	must be FALSE, the old fst file format is deprecated and can only be read and converted with fst package versions 0.8.0 to 0.8.10.

Value

Returns a list with meta information on the stored dataset in path. Has class fstmetadata.

Examples

```
# Sample dataset
x <- data.frame(
  First = 1:10,
  Second = sample(c(TRUE, FALSE, NA), 10, replace = TRUE),
  Last = sample(LETTERS, 10))

# Write to fst file
```

```
fst_file <- tempfile(fileext = ".fst")
write_fst(x, fst_file)

# Display meta information
metadata_fst(fst_file)
```

`threads_fst` *Get or set the number of threads used in parallel operations*

Description

For parallel operations, the performance is determined to a great extend by the number of threads used. More threads will allow the CPU to perform more computational intensive tasks simultaneously, speeding up the operation. Using more threads also introduces some overhead that will scale with the number of threads used. Therefore, using the maximum number of available threads is not always the fastest solution. With `threads_fst` the number of threads can be adjusted to the users specific requirements. As a default, `fst` uses a number of threads equal to the number of logical cores in the system.

Usage

```
threads_fst(nr_of_threads = NULL, reset_after_fork = NULL)
```

Arguments

`nr_of_threads`	number of threads to use or `NULL` to get the current number of threads used in multithreaded operations.
`reset_after_fork`	when `fst` is running in a forked process, the usage of OpenMP can create problems. To prevent these, `fst` switches back to single core usage when it detects a fork. After the fork, the number of threads is reset to it's initial setting. However, on some compilers (e.g. Intel), switching back to multi-threaded mode can lead to issues. When `reset_after_fork` is set to `FALSE`, `fst` is left in single-threaded mode after the fork ends. After the fork, multithreading can be activated again manually by calling `threads_fst` with an appropriate value for `nr_of_threads`. The default (`reset_after_fork = NULL`) leaves the fork behavior unchanged.

Details

The number of threads can also be set with `options(fst_threads = N)`. NOTE: This option is only read when the package's namespace is first loaded, with commands like `library`, `require`, or `::`. If you have already used one of these, you must use `threads_fst` to set the number of threads.

Value

the number of threads (previously) used

write_fst *Read and write fst files.*

Description

Read and write data frames from and to a fast-storage ('fst') file. Allows for compression and (file level) random access of stored data, even for compressed datasets. Multiple threads are used to obtain high (de-)serialization speeds but all background threads are re-joined before 'write_fst' and 'read_fst' return (reads and writes are stable). When using a 'data.table' object for 'x', the key (if any) is preserved, allowing storage of sorted data. Methods 'read_fst' and 'write_fst' are equivalent to 'read.fst' and 'write.fst' (but the former syntax is preferred).

Usage

```
write_fst(x, path, compress = 50, uniform_encoding = TRUE)

write.fst(x, path, compress = 50, uniform_encoding = TRUE)

read_fst(
  path,
  columns = NULL,
  from = 1,
  to = NULL,
  as.data.table = FALSE,
  old_format = FALSE
)

read.fst(
  path,
  columns = NULL,
  from = 1,
  to = NULL,
  as.data.table = FALSE,
  old_format = FALSE
)
```

Arguments

x	a data frame to write to disk
path	path to fst file
compress	value in the range 0 to 100, indicating the amount of compression to use. Lower values mean larger file sizes. The default compression is set to 50.
uniform_encoding	If 'TRUE', all character vectors will be assumed to have elements with equal encoding. The encoding (latin1, UTF8 or native) of the first non-NA element will used as encoding for the whole column. This will be a correct assumption

	for most use cases. If ‘uniform.encoding‘ is set to ‘FALSE‘, no such assumption will be made and all elements will be converted to the same encoding. The latter is a relatively expensive operation and will reduce write performance for character columns.
columns	Column names to read. The default is to read all columns.
from	Read data starting from this row number.
to	Read data up until this row number. The default is to read to the last row of the stored dataset.
as.data.table	If TRUE, the result will be returned as a data.table object. Any keys set on dataset x before writing will be retained. This allows for storage of sorted datasets. This option requires data.table package to be installed.
old_format	must be FALSE, the old fst file format is deprecated and can only be read and converted with fst package versions 0.8.0 to 0.8.10.

Value

‘read_fst‘ returns a data frame with the selected columns and rows. ‘write_fst‘ writes ‘x‘ to a ‘fst‘ file and invisibly returns ‘x‘ (so you can use this function in a pipeline).

Examples

```
# Sample dataset
x <- data.frame(A = 1:10000, B = sample(c(TRUE, FALSE, NA), 10000, replace = TRUE))

# Default compression
fst_file <- tempfile(fileext = ".fst")
write_fst(x, fst_file)  # filesize: 17 KB
y <- read_fst(fst_file) # read fst file
# Maximum compression
write_fst(x, fst_file, 100)  # fileSize: 4 KB
y <- read_fst(fst_file) # read fst file

# Random access
y <- read_fst(fst_file, "B") # read selection of columns
y <- read_fst(fst_file, "A", 100, 200) # read selection of columns and rows
```

Index