Package: MsBackendParquet (via r-universe)

November 25, 2026

Title Apache Parquet-based Mass Spectrometry Data Backend

Version 1.0.0

Description Mass spectrometry (MS) data backend for the Spectra Bioconductor package that stores and retrieves spectra and peak data from on-disk Apache Parquet datasets. Spectra metadata together with m/z and intensity values are stored in a columnar Parquet format that supports compression, dictionary encoding and Hive-style partitioning. Datasets are written with the Apache Arrow R bindings and queried with DuckDB. Spectra variables are held in a bounded in-memory index so that filtering and metadata access are fast while peak data remains on disk.

License Artistic-2.0

Encoding UTF-8

Depends R (>= 4.5.0), Spectra (>= 1.16.0)

Imports arrow (>= 14.0.0), BiocParallel, DBI, duckdb (>= 1.2.0), IRanges, jsonlite, methods, MsCoreUtils, ProtGenerics (>= 1.36.0), S4Vectors, stats, tools, utils, progress

Suggests BiocStyle, bench, dplyr, knitr, MsDataHub, MsExperiment, mzR, rmarkdown, SpectraQL, testthat (>= 3.0.0)

URL https://github.com/computational-metabolomics/MsBackendParquet

BugReports https://github.com/computational-metabolomics/MsBackendParquet/issues

biocViews Infrastructure, MassSpectrometry, Metabolomics, Proteomics, DataImport

VignetteBuilder knitr

Roxygen list(markdown = TRUE)

Collate 'MsBackendParquet-functions.R' 'MsBackendParquet.R' 'MsBackendParquet-creators.R' 'column-map.R' 'conditions.R' 'duckdb-connection.R' 'ingest.R' 'metadata-cache.R' 'mzml-activation.R' 'mzpeak-archive.R' 'mzstack-manifest.R' 'projections.R' 'sample-data.R' 'zzz.R'

Config/testthat/edition 3

Config/roxygen2/version 8.1.1

Config/pak/sysreqs cmake make libuv1-dev libxml2-dev libssl-dev

Repository Bioconductor 3.24

Date/Publication 2026-11-23 22:57:09 UTC

RemoteUrl https://github.com/bioc/MsBackendParquet

RemoteRef RELEASE_3_24

RemoteSha c1ebeaef543dd61b8a97c252197ff6e0f3881b60

Contents

buildProjection 2
createMsBackendParquetDataset 3
createMzPeakDataset 6
filterContainsMz 8
filterSampleData 9
MsBackendParquet 10
mzMLToParquet 15
mzstack-conditions 18
mzstack-manifest 19
runData 21

Index **23**

`buildProjection` *Build or drop signal projections*

Description

`buildProjection()` writes an m/z-ordered copy of a dataset's signal data, which makes queries that search for peaks at a given m/z across many runs substantially faster. `dropProjection()` deletes it again.

Usage

```
buildProjection(path, type = "mzsorted", runs = NULL, verbose = TRUE)

dropProjection(path, type = "mzsorted", runs = NULL, verbose = TRUE)
```

Arguments

path	`character(1)` with the dataset directory.
type	`character(1)` with the projection type. Currently only `"mzsorted"` is available.
runs	`character` with the run identifiers to build (or drop). The default, `NULL`, means every run in the dataset.
verbose	`logical(1)` whether to report progress.

Details

A projection is a **cache**: it is derived from the mzPeak archives, it can be deleted at any time, and no result depends on it for correctness. Queries that can use it do; queries that cannot fall back to reading the archives directly.

The cost is disk space: a projection is a second copy of the signal data, of broadly the same size (sorting compresses better, but how much better depends entirely on the data).

Value

`path`, invisibly.

Examples

```
## Projections are built from mzPeak datasets; two small archives ship
## with the package.
qc01 <- system.file("extdata", "QC01.mzpeak",
                    package = "MsBackendParquet")
path <- tempfile()
createMzPeakDataset(qc01, path = path, verbose = FALSE)
be <- backendInitialize(MsBackendParquet(), path = path)

buildProjection(path, verbose = FALSE)

## The query now reads a handful of m/z-ordered blocks instead of every
## spectrum's peaks. The answer is the same either way.
filterContainsMz(be, 201, ppm = 20)

## A projection is only a cache: dropping it frees the disk it used and
## changes nothing about the result.
dropProjection(path, verbose = FALSE)
filterContainsMz(be, 201, ppm = 20)
```

`createMsBackendParquetDataset`
Create a Parquet dataset from raw MS data or a `Spectra` *object*

Description

createMsBackendParquetDataset() imports raw MS data from one or more input files (mzML, mzXML or netCDF) into a new on-disk Parquet dataset that can be used by MsBackendParquet(). Alternatively, an existing DataFrame (such as the output of Spectra::spectraData()) can be written to disk directly via the data argument.

Usage

```
createMsBackendParquetDataset(
  path = character(),
  x = character(),
  data,
  backend = MsBackendMzR(),
  chunksize = 10L,
  partitioning = character(),
  compression = .DEFAULT_COMPRESSION,
  BPPARAM = BiocParallel::SerialParam(),
  engine = c("spectra", "mzr"),
  batch_size = 1000L,
  row_group_size = .DEFAULT_ROW_GROUP_SIZE
)
```

Arguments

path	character(1) with the path to the dataset directory. The directory must not already contain a MsBackendParquet dataset (use a fresh path for new datasets).
x	character with the names of the raw MS data files to import. Ignored if data is supplied.
data	Optional DataFrame with the full spectra data to be written. Must include the mz and intensity peaks variables. When supplied, x and backend are ignored.
backend	MS backend used to read the raw input files. Defaults to MsBackendMzR().
chunksize	integer(1) defining the number of input files processed per iteration.
partitioning	character with the names of the spectra variables to use as Hive partitioning columns below each run (e.g. "msLevel"). Use an empty vector (the default) to write a single un-partitioned set of files per run. "dataOrigin" is redundant whenever each source file is already its own run, which is always the case for x, and is then dropped with a warning.
compression	character(1) with the Parquet compression codec to use. Defaults to "snappy". See arrow::write_parquet() for supported values.
BPPARAM	Parallel processing setup used for reading the raw input files (only honoured when engine = "spectra"). Defaults to BiocParallel::SerialParam().
engine	character(1) selecting the import engine. The default "spectra" reads each chunk of files through a Spectra::Spectra() object (high throughput, supports BPPARAM). "mzr" streams spectra directly from mzML / mzXML / netCDF through mzR::openMSfile() into Apache Arrow's incremental Parquet writer, bounding memory to roughly batch_size spectra of peaks at a time. The

streaming engine requires only raw input files (no `data` argument) and ignores `chunksize` / `BPPARAM`. `mzR` must be installed.

`batch_size` `integer(1)` number of spectra per write call when `engine = "mzr"`. Larger values amortise write overhead at the cost of higher transient memory; smaller values keep memory tighter. Independent of `row_group_size`. Ignored for `engine = "spectra"`.

`row_group_size` `integer(1)` number of spectra per Parquet row group. Smaller row groups give finer-grained pruning for narrow range queries (e.g. retention time) at the cost of more file metadata; larger groups compress better and reduce per-group scan setup. Defaults to 250.

Details

The function writes a single Parquet dataset (under `<path>/spectra`) containing the spectra metadata plus the m/z and intensity values as Parquet `list<double>` columns.

With `data`, only the columns it carries are written, and a core Spectra variable that is entirely `NA` is not written at all; the backend reports a core variable without a column as missing values. Other columns, including list columns of peak annotations, are written as given.

Each source file becomes its own run, written to `<path>/spectra/run_id=<id>/`. If `partitioning` is supplied (e.g. `partitioning = "msLevel"`), Apache Arrow adds a further Hive level below the run, which typically enables Arrow's partition pruning when filtering and can substantially speed up access to large datasets.

Value

Invisibly returns `path` (with any tilde expanded).

See Also

`mzMLToParquet()` for a one-call wrapper over raw mzML files, `MsBackendParquet()` to open the resulting dataset.

Examples

```
library(Spectra)

## Spectra data can be written to a Parquet dataset directly, without
## going through raw files: any `DataFrame` with `mz` and `intensity`
## peaks variables will do.
sd <- S4Vectors::DataFrame(
    msLevel = c(1L, 1L, 2L),
    rtime = c(1.0, 2.0, 3.0),
    dataOrigin = c("file-a", "file-a", "file-b"))
sd$mz <- IRanges::NumericList(c(100, 110), c(101, 111),
                              c(102, 112), compress = FALSE)
sd$intensity <- IRanges::NumericList(c(10, 20), c(11, 21),
                                     c(12, 22), compress = FALSE)

## Each `dataOrigin` becomes its own run directory, so a filter on it
```

```
## skips whole files rather than scanning them. `msLevel` adds a second
## Hive level below the run.
path <- tempfile()
createMsBackendParquetDataset(path = path, data = sd,
                              partitioning = "msLevel")

be <- backendInitialize(MsBackendParquet(), path = path)
be
spectraVariables(be)
peaksData(be)

## Raw files are imported with `x` instead of `data`.
if (requireNamespace("mzR", quietly = TRUE)) {
    fl <- system.file("extdata", "demo.mzML",
                      package = "MsBackendParquet")
    from_files <- tempfile()
    createMsBackendParquetDataset(path = from_files, x = fl)
    print(backendInitialize(MsBackendParquet(), path = from_files))
}
```

`createMzPeakDataset` *Build a queryable dataset from mzPeak archives*

Description

`createMzPeakDataset()` registers one or more HUPO-PSI mzPeak archives as a single queryable dataset and `addMzPeakArchives()` adds more to an existing one.

The archives are **not modified**, and by default not copied: only a small derived index of the spectrum metadata is written, next to a manifest recording where each archive lives. Peak data continues to be read from the archives themselves.

Usage

```
createMzPeakDataset(
  archives,
  path,
  link = c("reference", "copy"),
  verbose = TRUE
)

addMzPeakArchives(
  path,
  archives,
  link = c("reference", "copy"),
  verbose = TRUE
)
```

Arguments

archives	character with paths to mzPeak archives. Each may be an unpacked directory or a .mzpeak ZIP file. ZIPs are unpacked into <path>/runs.
path	character(1) with the dataset directory. For createMzPeakDataset() it must not already hold a dataset.
link	character(1); "reference" (the default) records where each archive is and leaves it there, "copy" copies it under <path>/runs so the dataset is self-contained.
verbose	logical(1) whether to report progress.

Details

Ingest reads each archive's mzpeak_index.json, its spectrum metadata table and its scan / precursor / selected-ion side tables. It does not read signal data, so the cost is proportional to the number of spectra rather than to the size of the archives.

Side tables are flattened: a spectrum with several scans contributes the first, and the number that were present is recorded in n_scans (and likewise n_selected_ions, n_precursors). The full detail remains in the archive.

Archives using mzPeak's *chunked* signal layout are rejected; only the *point* layout is supported.

Value

path, invisibly.

See Also

MsBackendParquet() to open the resulting dataset.

Examples

```
library(Spectra)

## Two small mzPeak archives ship with the package.
qc01 <- system.file("extdata", "QC01.mzpeak",
                    package = "MsBackendParquet")
qc02 <- system.file("extdata", "QC02.mzpeak",
                    package = "MsBackendParquet")

path <- tempfile()
createMzPeakDataset(qc01, path = path, verbose = FALSE)

sps <- Spectra(backendInitialize(MsBackendParquet(), path = path))
length(sps)

## More runs can join the same dataset later; only the metadata index
## grows, the archives themselves are left where they are.
addMzPeakArchives(path, qc02, verbose = FALSE)

sps <- Spectra(backendInitialize(MsBackendParquet(), path = path))
```

```
length(sps)
table(dataOrigin(sps))
```

`filterContainsMz` *Select spectra containing a peak at a given m/z*

Description

`filterContainsMz()` keeps the spectra that hold at least one peak within a tolerance of any of the given m/z values. Unlike the peak-level filters in `Spectra`, the search happens inside the storage engine, so it does not read every spectrum's peaks into R.

Usage

```
filterContainsMz(object, mz = numeric(), tolerance = 0, ppm = 20)
```

Arguments

`object`	a `MsBackendParquet()` over an mzPeak dataset.
`mz`	`numeric` with the m/z values to look for.
`tolerance`	`numeric` with the absolute m/z tolerance.
`ppm`	`numeric` with the m/z-relative tolerance in parts per million.

Details

The query is much faster when the dataset has an `mzsorted` projection (see `buildProjection()`); without one it reads the archives directly and returns the same answer.

Value

`object` subset to the matching spectra, in their original order.

See Also

`buildProjection()` to make this query faster.

Examples

```
## A dataset over the two mzPeak archives shipped with the package. Their
## peaks sit in different mass ranges: QC01 near m/z 200, QC02 near 10100.
archives <- system.file("extdata", c("QC01.mzpeak", "QC02.mzpeak"),
                        package = "MsBackendParquet")
path <- tempfile()
createMzPeakDataset(archives, path = path, verbose = FALSE)
be <- backendInitialize(MsBackendParquet(), path = path)
length(be)

## Several m/z values can be searched at once; a spectrum is kept if it
```

```
## holds a peak near any of them.
res <- filterContainsMz(be, c(201, 10101), ppm = 20)
length(res)
peaksData(res)

## No match, no spectra.
filterContainsMz(be, 500)
```

`filterSampleData` *Select spectra by their run's sample metadata*

Description

`filterSampleData()` keeps the spectra whose run satisfies `expr`, an expression evaluated against the dataset's `runData()` table.

Because every run owns a contiguous block of `spectrum_id_`, the condition is resolved against the (small) sample metadata table in R and turned into a range predicate over those blocks. No per-spectrum column is read, and the predicate is carried forward so a subsequent `peaksData()` prunes Parquet row groups on the dataset's sort key.

Usage

```
filterSampleData(object, expr)
```

Arguments

`object`	`MsBackendParquet()` object.
`expr`	expression evaluated in the `runData()` table, returning a `logical` with one element per row. Variables not found there are looked up in the calling frame.

Value

`object` restricted to the matching spectra.

See Also

`runData()` to set the sample metadata being filtered on.

Examples

```
## See `?runData` for a worked example.
```

`MsBackendParquet` *Apache Parquet-based MS data backend*

Description

The `MsBackendParquet` is an implementation of `Spectra::MsBackend()` for `Spectra::Spectra()` objects that stores and retrieves mass spectrometry (MS) data from an on-disk **mzStack** dataset, built on Apache Parquet.

An mzStack dataset is a directory identified by an `mzStack.json` manifest. It holds one or more *runs*, of either kind:

- `native` - converted from raw MS data files by this package, with m/z and intensity stored as Parquet `list<double>` columns beside the spectra metadata;
- `mzpeak` - an index over external HUPO-PSI mzPeak archives, which are read where they are and never modified.

The manifest records which kind each run is, so a dataset never has to be identified by guessing from its directory layout.

The backend uses the Apache Arrow R bindings to read and write Parquet files. Spectra metadata together with the m/z and intensity values (stored as Parquet `list<double>` columns) live in one directory per run, which can optionally be further Hive-partitioned (for example by `msLevel`) to enable partition pruning.

New datasets can be created from raw MS data files using `createMsBackendParquetDataset()` or by changing the backend of an existing `Spectra` object with `Spectra::setBackend()`.

A backend can also be opened over a collection of HUPO-PSI mzPeak archives registered with `createMzPeakDataset()`. In that case the archives are read where they are and are never modified: the dataset holds only a small derived index of the spectrum metadata, and peak data is read from the archives on demand. See `createMzPeakDataset()` and `filterContainsMz()`.

See `vignette("MsBackendParquet")` for a worked introduction.

Usage

```
MsBackendParquet()

## S4 method for signature 'MsBackendParquet'
show(object)

## S4 method for signature 'MsBackendParquet'
backendInitialize(
  object,
  path = character(),
  data,
  representation = c("auto", "profile", "centroid"),
  ...
)
```

```
## S4 method for signature 'MsBackendParquet'
dataStorage(object)

## S4 method for signature 'MsBackendParquet'
x[i, j, ..., drop = FALSE]

## S4 method for signature 'MsBackendParquet,ANY'
extractByIndex(object, i)

## S4 method for signature 'MsBackendParquet'
peaksData(object, columns = c("mz", "intensity"))

## S4 method for signature 'MsBackendParquet'
peaksVariables(object)

## S4 method for signature 'MsBackendParquet'
intensity(object)

## S4 replacement method for signature 'MsBackendParquet'
intensity(object) <- value

## S4 method for signature 'MsBackendParquet'
mz(object)

## S4 replacement method for signature 'MsBackendParquet'
mz(object) <- value

## S4 replacement method for signature 'MsBackendParquet'
x$name <- value

## S4 method for signature 'MsBackendParquet'
spectraData(object, columns = spectraVariables(object))

## S4 method for signature 'MsBackendParquet'
reset(object)

## S4 method for signature 'MsBackendParquet'
spectraNames(object)

## S4 replacement method for signature 'MsBackendParquet'
spectraNames(object) <- value

## S4 method for signature 'MsBackendParquet'
filterMsLevel(object, msLevel = uniqueMsLevels(object))

## S4 method for signature 'MsBackendParquet'
filterRt(object, rt = numeric(), msLevel. = integer())
```

```
## S4 method for signature 'MsBackendParquet'
filterDataOrigin(object, dataOrigin = character())

## S4 method for signature 'MsBackendParquet'
filterPrecursorMzRange(object, mz = numeric())

## S4 method for signature 'MsBackendParquet'
filterPrecursorMzValues(object, mz = numeric(), ppm = 20, tolerance = 0)

## S4 method for signature 'MsBackendParquet'
uniqueMsLevels(object, ...)

## S4 method for signature 'MsBackendParquet'
backendMerge(object, ...)

## S4 method for signature 'MsBackendParquet'
precScanNum(object)

## S4 method for signature 'MsBackendParquet'
centroided(object)

## S4 method for signature 'MsBackendParquet'
smoothed(object)

## S4 method for signature 'MsBackendParquet'
tic(object, initial = TRUE)

## S4 method for signature 'MsBackendParquet'
supportsSetBackend(object, ...)

## S4 method for signature 'MsBackendParquet'
backendBpparam(object, BPPARAM = bpparam())
```

Arguments

object	A MsBackendParquet instance.
path	character(1) with the path to the Parquet dataset directory.
data	For backendInitialize(): optional DataFrame with the spectra data that should be written to a new Parquet dataset.
representation	For backendInitialize() on a dataset built from mzPeak archives: character(1) selecting which signal to read. mzPeak keeps profile and centroid data in separate files, and an archive may hold both. "auto" (the default) prefers centroids and falls back to profile data; "profile" and "centroid" ask for one specifically and fail if the archive does not have it. Ignored for datasets written by createMsBackendParquetDataset().
...	Additional arguments. For backendInitialize() with data: passed to createMsBackendParquetData

x	A MsBackendParquet instance (or a character with file paths for createMsBackendParquetDataset()
i	For [: integer, logical or character to subset the object.
j	For [: ignored.
drop	For [: ignored.
columns	For spectraData(): character with the names of the spectra variables to return. Defaults to all available variables. For peaksData(): character with the peaks variables to return.
value	Replacement value.
name	For $<-: character(1) with the variable name.
msLevel	For filterMsLevel(): integer with the MS levels to keep. For uniqueMsLevels(): ignored.
rt	For filterRt(): numeric(2) with the retention time range.
msLevel.	For filterRt(): integer with the MS levels on which the retention time filter should be applied.
dataOrigin	For filterDataOrigin(): character with the data origin values to keep.
mz	For filterPrecursorMzRange(): numeric(2) with the lower and upper precursor m/z bounds. For filterPrecursorMzValues(): numeric with the m/z values to keep.
ppm	For filterPrecursorMzValues(): numeric with the m/z-relative tolerance in parts-per-million.
tolerance	For filterPrecursorMzValues(): numeric with the absolute m/z tolerance.
initial	For tic(): logical(1) whether the original total ion current should be returned (TRUE, default) or computed from the intensities (FALSE).
BPPARAM	for backendBpparam(): parallel processing setup.

Details

The MsBackendParquet is principally a *read-only* backend; m/z and intensity values cannot be modified once they have been written to disk. However, by extending Spectra::MsBackendCached(), the backend allows users to *cache* additional or modified spectra variables locally within the object without rewriting the on-disk dataset.

Because the backend stores only a file system path (and not a live connection), MsBackendParquet objects can be serialised to disk with save() / base::saveRDS() and reused across parallel workers. The DuckDB connection used to read the dataset is held at package level, not on the object, and is keyed by process id, so a worker started in a *new* process opens its own.

Forked workers are the exception, and Spectra::backendBpparam() therefore downgrades a fork-based setup such as BiocParallel::MulticoreParam() to BiocParallel::SerialParam(). A fork inherits the parent's DuckDB connection as a live R external pointer whose finalizer, when the child next garbage collects, destroys the inherited database and waits for DuckDB's scheduler threads to join. fork() clones only the calling thread, so those threads do not exist in the child and the wait never completes. Cluster-based setups such as BiocParallel::SnowParam() start fresh processes, inherit nothing and are passed through unchanged. Little is given up by this: DuckDB parallelises scans internally, so the work is already spread across cores before Spectra chunks anything.

Value

See the description of the individual methods.

Creation of backend objects

New backend objects can be created with the `MsBackendParquet()` constructor. To attach the backend to an existing Parquet dataset (created previously with `createMsBackendParquetDataset()` or `backendInitialize()` with the `data` argument) use:

```
be <- backendInitialize(MsBackendParquet(), path = "/path/to/dataset")
```

- `createMsBackendParquetDataset()`: import raw MS data files into a new Parquet dataset. See the dedicated help page for details.
- `backendInitialize()`: attach the backend to an existing dataset. With the optional `data` argument (a `DataFrame` such as returned by `Spectra::spectraData()`), a **new** dataset is created at `path` from in-memory data. The dataset must not already exist. Additional arguments such as `partitioning` or `compression` are forwarded to `createMsBackendParquetDataset()`.
- `supportsSetBackend()`: returns `TRUE`. Changing to a `MsBackendParquet` is supported through `Spectra::setBackend()`, passing the destination `path` (and optional `partitioning` / `compression`) as named arguments.

Subsetting, merging and filtering data

`MsBackendParquet` objects can be subsetted with `[` or `extractByIndex()`. Internally this only subsets the in-memory vector of primary keys; the on-disk dataset is never modified. Any subsetting can be undone with `reset()`.

Multiple `MsBackendParquet` objects pointing to the **same** dataset path can be combined with `backendMerge()` (mostly useful after a `split()`).

Filter methods optimised for `MsBackendParquet` (via Arrow's predicate push-down) are:

- `filterDataOrigin()`: keep spectra with matching `dataOrigin`.
- `filterMsLevel()`: keep spectra with the requested MS levels.
- `filterPrecursorMzRange()`: keep spectra with `precursorMz` in the given range.
- `filterPrecursorMzValues()`: keep spectra with `precursorMz` matching one of the provided values (with `ppm` / `tolerance`).
- `filterRt()`: keep spectra with retention times in the given range, optionally restricted to selected MS levels.

If a filter variable has been modified locally with `$<-` the default `Spectra::MsBackendCached()` implementation is used instead so that the cached values take precedence over the on-disk ones.

Accessing data

- `dataStorage()`: returns a `character` vector with the dataset path, one element per spectrum.
- `intensity()` / `mz()`: return `IRanges::NumericList()` objects.
- `intensity<-` / `mz<-`: not supported.

- `centroided()` / `smoothed()`: return a `logical` vector of length `length(object)`, `NA` where the source data does not say.
- `peaksData()`: returns a `list` of two-column matrices.
- `peaksVariables()`: returns `c("mz", "intensity")`, followed by any peak-annotation variable a native dataset stores beside them: a list column with one value per peak, such as a merged spectrum's per-peak signal-to-noise. Such variables are written by passing them as list columns in `data` to `createMsBackendParquetDataset()`.
- `precScanNum()`: returns an `integer` vector of length `length(object)` with the scan number of each spectrum's precursor, `NA` for spectra without one.
- `reset()`: re-initialises the backend from disk and drops cached spectra variables.
- `spectraData()`: returns a `DataFrame` with the requested spectra variables. Locally cached variables are merged with on-disk ones.
- `spectraNames()`: returns the (character) `spectrum_id_` values.
- `tic()`: returns the original `totIonCurrent` (for `initial = TRUE`) or computes it from the intensity values.
- `uniqueMsLevels()`: returns the unique MS levels in the dataset.

Examples

```
library(MsBackendParquet)

# Create a tiny in-memory `Spectra`-like data frame.
sd <- S4Vectors::DataFrame(
    msLevel = c(1L, 1L, 2L),
    rtime = c(1.0, 2.0, 3.0),
    dataOrigin = "memory")
sd$mz <- IRanges::NumericList(c(100, 110), c(101, 111),
                              c(102, 112), compress = FALSE)
sd$intensity <- IRanges::NumericList(c(10, 20), c(11, 21),
                                     c(12, 22), compress = FALSE)

path <- tempfile()
be <- backendInitialize(MsBackendParquet(), path = path, data = sd)
be
spectraVariables(be)
peaksData(be)
```

`mzMLToParquet` *Convert mzML (or related) files into a Parquet dataset*

Description

`mzMLToParquet()` is a thin convenience wrapper around `createMsBackendParquetDataset()` tailored to the most common use-case: importing one or more mass spectrometry data files (mzML, mzXML, netCDF) from disk into a new Apache Parquet dataset that can be read back by `MsBackendParquet()`.

Compared to calling `createMsBackendParquetDataset()` directly, this function performs basic input validation (existence and recognised file extension), optionally tears down a stale destination directory, prints a short summary of what was written and returns an initialised `MsBackendParquet` instance ready to be wrapped in a `Spectra::Spectra()` object.

Usage

```
mzMLToParquet(
  files,
  path,
  partitioning = character(),
  compression = .DEFAULT_COMPRESSION,
  chunksize = 10L,
  overwrite = FALSE,
  BPPARAM = BiocParallel::SerialParam(),
  engine = c("spectra", "mzr"),
  batch_size = 1000L,
  row_group_size = .DEFAULT_ROW_GROUP_SIZE,
  verbose = TRUE
)
```

Arguments

`files`	`character` with the paths to the input mzML, mzXML or netCDF files. Files must exist on disk and have a recognised extension (`.mzML`, `.mzXML`, `.cdf`, `.nc`; case-insensitive). mzML and mzXML may be gzip-compressed (`.mzML.gz`, `.mzXML.gz`). Each file becomes one run, named after the file without its extension and compression suffix.
`path`	`character(1)` with the destination directory for the Parquet dataset. The directory must not already contain a `MsBackendParquet` dataset, unless `overwrite = TRUE`.
`partitioning`	`character` with the spectra variables to use as Hive partitioning columns below each run (e.g. `"msLevel"`). Empty (default) writes a single un-partitioned set of files per run. `"dataOrigin"` is redundant, since each input file is already its own run, and is dropped with a warning.
`compression`	`character(1)` Parquet compression codec, passed to `arrow::write_parquet()` / `arrow::write_dataset()`. Defaults to `"snappy"`.
`chunksize`	`integer(1)` number of input files processed per iteration. Higher values trade memory for throughput.
`overwrite`	`logical(1)` whether to remove an existing dataset at `path` before writing. Defaults to `FALSE`.
`BPPARAM`	Parallel processing setup forwarded to the underlying `Spectra::Spectra()` / `MsBackendMzR` reads (only honoured when `engine = "spectra"`).
`engine`	`character(1)` import engine, passed through to `createMsBackendParquetDataset()`. `"spectra"` (default) goes through `Spectra::Spectra()` and `MsBackendMzR()`; `"mzr"` streams batches of spectra directly from mzML via `mzR::openMSfile()` for bounded-memory imports of large files. `mzR` must be installed for `"mzr"`.

batch_size	integer(1) number of spectra per write batch when engine = "mzr". Ignored otherwise.
row_group_size	integer(1) number of spectra per Parquet row group. See createMsBackendParquetDataset() for details. Defaults to 250.
verbose	logical(1) whether to print a short summary of the resulting dataset.

Value

Invisibly returns an initialised MsBackendParquet() instance pointing at the new dataset.

Spectra variables

Both engines store the same spectra variables, those MsBackendMzR reports for the file, with the same values. scanIndex is the spectrum's 1-based position in its file, acquisitionNum the instrument's scan number, and precScanNum the acquisitionNum of the precursor spectrum, so an MSn spectrum's parent is the spectrum of the same run whose acquisitionNum equals its precScanNum.

activationType holds the dissociation method of a spectrum's first precursor (its own, for an MSn spectrum) as a PSI-MS accession, for example "MS:1000422" for beam-type collision-induced dissociation (HCD) or "MS:1000133" for collision-induced dissociation. mzR does not report it, so it is read from the mzML <activation> elements in a second pass over the file. It is missing where the file records none, and for mzXML and netCDF input.

See Also

createMsBackendParquetDataset() for the full API, MsBackendParquet() for the backend class.

Examples

```
library(Spectra)

## Reading mzML needs `mzR`.
if (requireNamespace("mzR", quietly = TRUE)) {
    files <- system.file("extdata", "demo.mzML",
                         package = "MsBackendParquet")

    ## Convert and open in one call: the returned backend is already
    ## initialized on the new dataset.
    be <- mzMLToParquet(files, path = tempfile(),
                        partitioning = "msLevel", verbose = FALSE)
    print(be)

    sps <- Spectra(be)
    print(rtime(sps))
    print(peaksData(sps)[[1L]])
}
```

`mzstack-conditions` *Typed mzStack error conditions*

Description

The mzStack specification (mzStack-5 §3) requires an implementation to distinguish a fixed set of failure classes. The class names and the mechanism are left to each implementation; the distinctions are not. In R they are condition classes, so a caller can react to one kind of failure without matching message text:

```
tryCatch(Spectra(backendInitialize(MsBackendParquet(), path = p)),
         mzstack_format = function(e) NULL)
```

Every condition inherits from `mzstack_error`, then `error` and `condition`, so existing `tryCatch(..., error = )` handlers keep working.

Class	Raised when
`mzstack_format`	the directory is not readable as mzStack: no manifest, another format, an unsupported major versior
`mzstack_archive`	a referenced mzPeak archive cannot be read as its specification requires
`mzstack_reference`	a source dataset another dataset declares cannot be located, cannot be read, or has a different `uid`
`mzstack_study`	a repository study cannot be ingested
`mzstack_stale`	a projection, index or reference no longer matches the run it was built from
`mzstack_unsupported`	a valid request outside what this implementation supports
`mzstack_semantic`	a well-formed request that is incoherent
`mzstack_capability`	a request for data the dataset does not contain
`mzstack_resource`	a request that cannot be completed within a declared bound

A `mzstack_capability` error and an empty result are different outcomes: the first says the question could not be asked, the second that nothing matched.

One further class is not a failure the specification classifies but a property of concurrent writing: `mzstack_conflict` is raised by `writeManifest()` when another writer holds the manifest lock or has committed since the manifest was read. Nothing has been written; re-read the manifest and re-apply the change.

`mzstackError()` is exported so that packages layered on MsBackendParquet raise the same classes.

Usage

```
mzstackCondition(
  class = .MZSTACK_ERROR_CLASSES,
  ...,
  data = list(),
  call = NULL
)

mzstackError(class = .MZSTACK_ERROR_CLASSES, ..., data = list(), call = NULL)
```

Arguments

class	character(1), one of the classes above without the mzstack_ prefix.
...	pieces of the message, pasted together as by stop().
data	named list of further fields to carry on the condition, for example the run or source key the error concerns.
call	the call to record on the condition. Defaults to none, which matches the call. = FALSE convention used throughout the package.

Value

mzstackError() does not return; it signals the condition. mzstackCondition() returns it unsignalled.

Examples

```
res <- tryCatch(mzstackError("capability", "no such table"),
                mzstack_capability = function(e) conditionMessage(e))
res
```

---

mzstack-manifest *Read and write an mzStack manifest*

Description

Every mzStack dataset is described by a manifest, mzStack.json, at the root of its directory. These functions give packages layered on MsBackendParquet, such as a results layer that adds its own top-level keys, access to it without re-implementing the format's rules:

- newManifest() returns the manifest of an empty dataset: no runs, at generation 1.
- readManifest() parses and validates the manifest. Keys this package does not interpret are returned exactly as parsed, arrays as lists, so they survive a read/write cycle unchanged.
- writeManifest() replaces the manifest atomically: the new version is written to a temporary file and renamed into place, so a reader never observes a half-written manifest. It increments generation, refuses to overwrite a manifest another writer has committed since manifest was read, and drops every cache this package holds for the dataset. The check and the replacement are serialised by a lock directory, mzStack.json.lock; a held lock or a changed generation raises a mzstack_conflict error and writes nothing. Numbers are written so that they read back exactly. Where path holds no manifest yet, the directory is created if needed and manifest is written as the dataset's first, without incrementing generation.
- manifestRuns() returns the run entries as a data.frame, one row per run: run_id, kind, path, n_spectra, uid_base, ingested_at, layout, profile, centroid and source. The tuple (run_id, uid_base, n_spectra, ingested_at) is the run's fingerprint: while it is unchanged, the run's spectrum_id_ values address the same spectra.
- invalidateDatasetCache() drops the parsed manifest, the DuckDB view, the metadata cache and the sample metadata this process holds for a dataset. writeManifest() calls it; call it directly after changing a dataset's files by other means. A manifest replaced by another process is detected without it.

The manifest must be written last, after every file it declares is durably on disk: its atomic replacement is the commit point of any write.

Usage

```
newManifest()

readManifest(path)

writeManifest(path, manifest, bump = TRUE)

manifestRuns(path)

invalidateDatasetCache(path)
```

Arguments

path	character(1), the dataset directory, or an MsBackendParquet over it.
manifest	list, as returned by readManifest() and modified.
bump	logical(1), whether to increment generation. Every change to a dataset's content must, so leave this TRUE unless the change is one the specification exempts.

Value

newManifest() and readManifest() return a list. writeManifest() returns the written manifest invisibly, with its new generation. manifestRuns() returns a data.frame. invalidateDatasetCache() returns NULL invisibly.

Examples

```
fl <- system.file("extdata", "QC01.mzpeak", package = "MsBackendParquet")
ds <- createMzPeakDataset(fl, file.path(tempdir(), "manifest-example"))
m <- readManifest(ds)
m$generation
manifestRuns(ds)[, c("run_id", "uid_base", "n_spectra", "ingested_at")]

## Add a key of another layer. It survives this package's own writes.
m$role <- "study"
m <- writeManifest(ds, m)
m$generation
```

runData *Per-run sample metadata*

Description

`runData()` reads, and `runData<-()` writes, a dataset's per-run sample metadata: one row per run, keyed by `run_id`, holding whatever the experiment records about the sample that run came from.

A run is one MS run – one mzPeak archive, or one converted source file – and owns a contiguous block of the dataset's `spectrum_id_`, so the sample metadata's columns become ordinary spectra variables and `filterSampleData()` can select on them by id range rather than by reading a per-spectrum column.

The sample metadata is written to `index/samples.parquet` and can be replaced at any time after ingest. Replacing it does not invalidate projections: it does not touch the signal.

`runVariables()` lists the sample metadata columns.

Usage

```
runData(x)

runVariables(x)

runData(x) <- value
```

Arguments

`x`	dataset path, or an `MsBackendParquet()` opened on one.
`value`	`data.frame` with a `run_id` column naming runs of the dataset, plus one column per sample metadata variable.

Details

Column names are checked against the dataset's existing spectra variables and against the mzPeak column vocabulary. A sample metadata column called `time` or `id` would otherwise silently shadow the source of a core variable, so it is refused rather than accepted and later mis-read.

Runs with no row in the table yield `NA`, so metadata covering only some of a dataset's runs is legal.

Value

`runData()` a `data.frame` with one row per described run; `runVariables()` a `character` of sample metadata column names.

See Also

`filterSampleData()` to select spectra on it.

Examples

```
library(Spectra)

sd <- S4Vectors::DataFrame(
    msLevel = c(1L, 1L, 2L),
    rtime = c(1.0, 2.0, 3.0),
    dataOrigin = c("QC01", "QC01", "QC02"))
sd$mz <- IRanges::NumericList(c(100, 110), c(101, 111),
                              c(102, 112), compress = FALSE)
sd$intensity <- IRanges::NumericList(c(10, 20), c(11, 21),
                                     c(12, 22), compress = FALSE)
path <- tempfile()
createMsBackendParquetDataset(path = path, data = sd)

## One row per run; `run_id` is the source file's name.
runData(path) <- data.frame(run_id = c("QC01", "QC02"),
                            subject = c("S3", "S3"),
                            timepoint = c(0, 6))

be <- backendInitialize(MsBackendParquet(), path = path)
runVariables(be)
spectraData(be, c("rtime", "timepoint"))

## Selecting on it is a range query over the runs' id blocks.
filterSampleData(be, timepoint == 6)
```

Index

[,MsBackendParquet-method
(MsBackendParquet), 10
$<-,MsBackendParquet-method
(MsBackendParquet), 10

addMzPeakArchives
(createMzPeakDataset), 6
arrow::write_dataset(), *16*
arrow::write_parquet(), *4*, *16*

backendBpparam,MsBackendParquet-method
(MsBackendParquet), 10
backendInitialize,MsBackendParquet-method
(MsBackendParquet), 10
backendMerge,MsBackendParquet-method
(MsBackendParquet), 10
base::saveRDS(), *13*
BiocParallel::MulticoreParam(), *13*
BiocParallel::SerialParam(), *4*, *13*
BiocParallel::SnowParam(), *13*
buildProjection, 2
buildProjection(), *8*

centroided,MsBackendParquet-method
(MsBackendParquet), 10
createMsBackendParquetDataset, 3
createMsBackendParquetDataset(), *10*, *12*, *14–17*
createMzPeakDataset, 6
createMzPeakDataset(), *10*

dataStorage,MsBackendParquet-method
(MsBackendParquet), 10
dropProjection (buildProjection), 2

extractByIndex,MsBackendParquet,ANY-method
(MsBackendParquet), 10

filterContainsMz, 8
filterContainsMz(), *10*
filterDataOrigin,MsBackendParquet-method
(MsBackendParquet), 10
filterMsLevel,MsBackendParquet-method
(MsBackendParquet), 10
filterPrecursorMzRange,MsBackendParquet-method
(MsBackendParquet), 10
filterPrecursorMzValues,MsBackendParquet-method
(MsBackendParquet), 10
filterRt,MsBackendParquet-method
(MsBackendParquet), 10
filterSampleData, 9
filterSampleData(), *21*

intensity,MsBackendParquet-method
(MsBackendParquet), 10
intensity<-,MsBackendParquet-method
(MsBackendParquet), 10
invalidateDatasetCache
(mzstack-manifest), 19
IRanges::NumericList(), *14*

manifestRuns (mzstack-manifest), 19
MsBackendParquet, 10
MsBackendParquet(), *4*, *5*, *7–9*, *15*, *17*, *21*
MsBackendParquet-class
(MsBackendParquet), 10
MsBackendParquet-package
(MsBackendParquet), 10
mz,MsBackendParquet-method
(MsBackendParquet), 10
mz<-,MsBackendParquet-method
(MsBackendParquet), 10
mzMLToParquet, 15
mzMLToParquet(), *5*
mzR::openMSfile(), *4*, *16*
mzstack-conditions, 18
mzstack-manifest, 19
mzstackCondition (mzstack-conditions), 18
mzstackError (mzstack-conditions), 18

newManifest *(mzstack-manifest)*, 19

peaksData,MsBackendParquet-method
(MsBackendParquet), 10
peaksVariables,MsBackendParquet-method
(MsBackendParquet), 10
precScanNum,MsBackendParquet-method
(MsBackendParquet), 10

readManifest *(mzstack-manifest)*, 19
reset,MsBackendParquet-method
(MsBackendParquet), 10
runData, 21
runData(), *9*
runData<- *(runData)*, 21
runVariables *(runData)*, 21

save(), *13*
show,MsBackendParquet-method
(MsBackendParquet), 10
smoothed,MsBackendParquet-method
(MsBackendParquet), 10
Spectra::backendBpparam(), *13*
Spectra::MsBackend(), *10*
Spectra::MsBackendCached(), *13, 14*
Spectra::setBackend(), *10*, *14*
Spectra::Spectra(), *4*, *10*, *16*
Spectra::spectraData(), *4*, *14*
spectraData,MsBackendParquet-method
(MsBackendParquet), 10
spectraNames,MsBackendParquet-method
(MsBackendParquet), 10
spectraNames<-,MsBackendParquet-method
(MsBackendParquet), 10
stop(), *19*
supportsSetBackend,MsBackendParquet-method
(MsBackendParquet), 10

tic,MsBackendParquet-method
(MsBackendParquet), 10

uniqueMsLevels,MsBackendParquet-method
(MsBackendParquet), 10

writeManifest *(mzstack-manifest)*, 19
writeManifest(), *18*