Package: RBCFTools (via r-universe)

June 8, 2026

Title 'BCFTools', 'libbcftools' and 'htslib' Wrappers and 'BCF'/'VCF' to 'Parquet' Convertors

Version 1.23-0.0.3.1.9001

Description Bundles the 'htslib' and 'bcftools' libraries and command lines tools for reading and manipulating VCF/BCF files. Includes streaming facilities from VCF to Apache Arrow via 'nanoarrow', enabling export to Arrow IPC format and Parquet format using 'duckdb' including a 'bcf_reader' extension. Utilities for reading and writing VCF/BCF files into 'DuckLake' are included. provided.

License GPL (>= 3)

Copyright See inst/AUTHORS

Depends R (>= 4.4.0), parallel

Imports nanoarrow, vctrs

LinkingTo nanoarrow

SystemRequirements GNU make, pkg-config, libcurl, libgsl, libdeflate, libbzip2, libzlib, libssl-dev (or other ssl library), liblzma, libsuitesparse-dev/libcholmod5 (optional for pgs plugin)

Suggests tinytest, vcfppR, duckdb, processx, DBI

OS_type unix

URL https://github.com/RGenomicsETL/RBCFTools, https://rgenomicsetl.github.io/RBCFTools/

BugReports https://github.com/RGenomicsETL/RBCFTools/issues

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Config/pak/sysreqs make libgsl0-dev liblzma-dev libzstd-dev pkg-config

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

Date/Publication 2026-04-09 15:32:14 UTC

RemoteUrl https://github.com/RGenomicsETL/RBCFTools

RemoteRef HEAD

RemoteSha 6ee59bc75ad79dee265f8ac3594f825ba3cb8ae8

Contents

annot_tsv_path *Get Path to annot-tsv Executable*

Description

Returns the path to the bundled annot-tsv executable.

Usage

```
annot_tsv_path()
```

Value

A character string containing the path to the annot-tsv executable.

Examples

```
annot_tsv_path()
```

bcf_reader_build *Build the bcf_reader DuckDB extension*

Description

Compiles the bcf_reader extension from source using the package's htslib. Source files are copied to the build directory first.

Usage

```
bcf_reader_build(build_dir, force = FALSE, verbose = TRUE)
```

Arguments

build_dir	Directory where to build the extension. Source files will be copied here and the extension will be built in build_dir/build/.
force	Logical, force rebuild even if extension exists
verbose	Logical, show build output

Value

Path to the built extension file

Examples

```
## Not run:
# Build in temp directory
ext_path <- bcf_reader_build(tempdir())

# Build in a specific location
ext_path <- bcf_reader_build("/tmp/bcf_reader")

# Force rebuild
ext_path <- bcf_reader_build("/tmp/bcf_reader", force = TRUE)

## End(Not run)
```

`bcf_reader_copy_source` *Copy bcf_reader extension source to a build directory*

Description

Copies the extension source files from the package to a specified directory for building. This is necessary because the installed package directory is typically read-only.

Usage

```
bcf_reader_copy_source(dest_dir)
```

Arguments

`dest_dir`	Directory where to copy the source files.

Value

Invisible path to the destination directory

Examples

```
## Not run:
# Copy to temp directory
build_dir <- bcf_reader_copy_source(tempdir())

# Copy to a specific location
build_dir <- bcf_reader_copy_source("/tmp/bcf_reader_build")

## End(Not run)
```

`bcftools_bin_dir` *Get Path to bcftools Binary Directory*

Description

Returns the path to the directory containing bcftools and related scripts.

Usage

```
bcftools_bin_dir()
```

Details

The directory contains the following tools:

- `bcftools` - Main bcftools executable
- `color-chrs.pl` - Chromosome coloring script
- `gff2gff` - GFF conversion tool
- `gff2gff.py` - GFF conversion Python script
- `guess-ploidy.py` - Ploidy guessing script
- `plot-roh.py` - ROH plotting script
- `plot-vcfstats` - VCF statistics plotting script
- `roh-viz` - ROH visualization tool
- `run-roh.pl` - ROH analysis script
- `vcfutils.pl` - VCF utilities script
- `vrfs-variances` - Variant frequency variances tool

Value

A character string containing the path to the bcftools bin directory.

Examples

```
bcftools_bin_dir()
```

bcftools_lib_dir *Get bcftools Library Directory*

Description

Returns the path to the bcftools library files for use in linking.

Usage

```
bcftools_lib_dir()
```

Details

This directory contains `libbcftools.a` (static) and `libbcftools.so` (shared) libraries.

Value

A character string containing the path to the bcftools lib directory.

Examples

```
bcftools_lib_dir()
```

bcftools_libs *Get Linker Flags for bcftools Library*

Description

Returns the linker flags needed to link against the bcftools library.

Usage

```
bcftools_libs()
```

Details

Note that bcftools library also depends on htslib, so you typically need to include both `bcftools_libs()` and `htslib_libs()` in your linker flags.

Value

A character string containing linker flags including `-L` library path and `-l` library name.

Examples

```
bcftools_libs()
# Full linking: paste(RBCFTools::bcftools_libs(), RBCFTools::htslib_libs())
```

bcftools_path *Get Path to bcftools Executable*

Description

Returns the path to the bundled bcftools executable.

Usage

```
bcftools_path()
```

Value

A character string containing the path to the bcftools executable.

Examples

```
bcftools_path()
```

bcftools_plugins_dir *Get Path to bcftools Plugins Directory*

Description

Returns the path to the directory containing bcftools plugins.

Usage

```
bcftools_plugins_dir()
```

Value

A character string containing the path to the bcftools plugins directory.

Examples

```
bcftools_plugins_dir()
```

bcftools_tools *List Available bcftools Scripts*

Description

Lists all available scripts and tools in the bcftools bin directory.

Usage

```
bcftools_tools()
```

Value

A character vector of available tool names.

Examples

```
bcftools_tools()
```

bcftools_version *Get bcftools Version*

Description

Returns the version string of the bundled bcftools library.

Usage

```
bcftools_version()
```

Value

A character string containing the bcftools version.

Examples

```
bcftools_version()
```

bgzip_path *Get Path to bgzip Executable*

Description

Returns the path to the bundled bgzip executable.

Usage

```
bgzip_path()
```

Value

A character string containing the path to the bgzip executable.

Examples

```
bgzip_path()
```

ducklake *DuckLake helpers for VCF/BCF ETL*

Description

Utilities to load the DuckLake extension, attach a lake (local or S3-backed), configure S3 secrets, and write variants using either direct DuckDB insert or parallel Parquet conversion.

ducklake_attach *Attach a DuckLake catalog (legacy function)*

Description

Attach a DuckLake catalog (legacy function)

Usage

```
ducklake_attach(
  con,
  metadata_path,
  data_path,
  alias = "ducklake",
  read_only = FALSE,
  create_if_missing = TRUE,
  extra_options = list()
)
```

Arguments

con	A DuckDB connection with DuckLake loaded.
metadata_path	Path/URI to the DuckLake metadata DB (without the ducklake: prefix).
data_path	Path/URI for table data (Parquet files).
alias	Schema alias to attach as. Default: "ducklake".
read_only	Logical, open lake read-only. Default: FALSE.
create_if_missing	Logical, create metadata DB if missing. Default: TRUE.
extra_options	Named list of additional ATTACH options (e.g., list(METADATA_CATALOG = "meta")).

Value

Invisible NULL.

See Also

ducklake_connect_catalog for abstracted backend support.

---

ducklake_connect_catalog

Connect to a DuckLake catalog with abstracted backend support

---

Description

Connect to a DuckLake catalog with abstracted backend support

Usage

```
ducklake_connect_catalog(
  con,
  backend = c("duckdb", "sqlite", "postgresql", "mysql"),
  connection_string = NULL,
  data_path = NULL,
  alias = "ducklake",
  secret_name = NULL,
  read_only = FALSE,
  create_if_missing = TRUE,
  extra_options = list()
)
```

Arguments

con	A DuckDB connection with DuckLake loaded.
backend	Database backend type ("duckdb", "sqlite", "postgresql", "mysql").
connection_string	Database connection string (format depends on backend).
data_path	Path/URI for table data (Parquet files). Required for new lakes.
alias	Schema alias to attach as. Default: "ducklake".
secret_name	Optional secret name to use instead of direct connection parameters.
read_only	Logical, open lake read-only. Default: FALSE.
create_if_missing	Logical, create metadata DB if missing. Default: TRUE.
extra_options	Named list of additional ATTACH options.

Value

Invisible NULL.

---

ducklake_create_catalog_secret

Create a DuckLake catalog secret for database credentials

Description

Create a DuckLake catalog secret for database credentials

Usage

```
ducklake_create_catalog_secret(
  con,
  name = "ducklake_catalog",
  backend = c("duckdb", "sqlite", "postgresql", "mysql"),
  connection_string,
  data_path = NULL,
  metadata_parameters = list(),
  persistent = FALSE
)
```

Arguments

con	A DuckDB connection.
name	Secret name (identifier). Default: "ducklake_catalog".
backend	Database backend type ("duckdb", "sqlite", "postgresql", "mysql").
connection_string	Database connection string (without ducklake: prefix).

data_path	Default data path for this catalog. Optional.
metadata_parameters	Named list of additional metadata parameters.
persistent	Logical, create a persistent secret. Default: FALSE.

Value

Invisible NULL.

`ducklake_create_s3_secret` *Create or replace an S3 secret for DuckLake*

Description

Create or replace an S3 secret for DuckLake

Usage

```
ducklake_create_s3_secret(
  con,
  name = "ducklake_s3",
  key_id,
  secret,
  endpoint = NULL,
  region = NULL,
  use_ssl = TRUE,
  url_style = "path",
  session_token = NULL
)
```

Arguments

con	A DuckDB connection.
name	Secret name (identifier). Default: "ducklake_s3".
key_id	S3 key ID.
secret	S3 secret key.
endpoint	Optional S3-compatible endpoint (e.g., "s3.us-east-1.amazonaws.com" or "minio:9000").
region	Optional region.
use_ssl	Logical, whether to use SSL. Default: TRUE.
url_style	URL style ("path" or "virtual_host"). Default: "path".
session_token	Optional session token.

Value

Invisible NULL.

ducklake_current_snapshot *Get current snapshot ID*

Description

Get current snapshot ID

Usage

```
ducklake_current_snapshot(con, catalog = "lake")
```

Arguments

con	DuckDB connection with DuckLake attached.
catalog	DuckLake catalog name.

Value

Integer snapshot ID.

ducklake_download_mc *Download a static MinIO client (mc) binary*

Description

Download a static MinIO client (mc) binary

Usage

```
ducklake_download_mc(dest_dir = tempdir(), url = NULL, filename = "mc")
```

Arguments

dest_dir	Destination directory (created if missing).
url	Optional download URL. Defaults to mc Linux build for host arch.
filename	Output filename. Defaults to "mc".

Value

Path to downloaded binary.

ducklake_download_minio *Download a static MinIO server binary*

Description

Download a static MinIO server binary

Usage

```
ducklake_download_minio(dest_dir = tempdir(), url = NULL, filename = "minio")
```

Arguments

dest_dir	Destination directory (created if missing).
url	Optional download URL. Defaults to MinIO Linux build for host arch.
filename	Output filename. Defaults to "minio".

Value

Path to downloaded binary.

ducklake_drop_secret *Drop a DuckLake catalog secret*

Description

Drop a DuckLake catalog secret

Usage

```
ducklake_drop_secret(con, name)
```

Arguments

con	A DuckDB connection.
name	Secret name to drop.

Value

Invisible NULL.

ducklake_list_files *List files managed by DuckLake for a table*

Description

List files managed by DuckLake for a table

Usage

```
ducklake_list_files(con, catalog = "lake", table, schema = "main")
```

Arguments

con	DuckDB connection with DuckLake attached.
catalog	DuckLake catalog name.
table	Table name.
schema	Schema name (default "main").

Value

Data frame with file information.

ducklake_list_secrets *List existing DuckLake catalog secrets*

Description

List existing DuckLake catalog secrets

Usage

```
ducklake_list_secrets(con)
```

Arguments

con	A DuckDB connection.

Value

Data frame with columns: name, type, metadata_path, data_path.

ducklake_load *Load the DuckLake extension*

Description

Load the DuckLake extension

Usage

```
ducklake_load(con, install = TRUE)
```

Arguments

con	A DuckDB connection.
install	Logical, attempt `INSTALL ducklake` before loading. Defaults to TRUE.

Value

The connection (invisibly).

ducklake_load_vcf *Load VCF into DuckLake (ETL + Registration)*

Description

Converts VCF/BCF to Parquet using the fast `bcf_reader` extension, then registers the Parquet file in a DuckLake catalog table.

Usage

```
ducklake_load_vcf(
  con,
  table,
  vcf_path,
  extension_path,
  output_path = NULL,
  threads = parallel::detectCores(),
  compression = "zstd",
  row_group_size = 100000L,
  region = NULL,
  columns = NULL,
  overwrite = FALSE,
  allow_evolution = FALSE,
  tidy_format = FALSE,
  partition_by = NULL
)
```

Arguments

con	DuckDB connection with DuckLake attached.
table	Target table name (optionally qualified, e.g., "lake.variants").
vcf_path	Path/URI to VCF/BCF file.
extension_path	Path to bcf_reader.duckdb_extension (required).
output_path	Optional Parquet output path. If NULL, uses DuckLake's DATA_PATH.
threads	Number of threads for conversion.
compression	Parquet compression codec.
row_group_size	Parquet row group size.
region	Optional region filter (e.g., "chr1:1000-2000").
columns	Optional character vector of columns to include.
overwrite	Logical, drop existing table first.
allow_evolution	Logical, evolve table schema by adding new columns from VCF. Default: FALSE. When TRUE, new columns found in the VCF are added via ALTER TABLE before insertion, making all columns queryable. Useful for combining VCFs with different annotations (e.g., VEP columns) or different samples (FORMAT_*_SampleName).
tidy_format	Logical, if TRUE exports data in tidy (long) format with one row per variant-sample combination and a SAMPLE_ID column. Default FALSE. Ideal for cohort analysis and combining multiple single-sample VCFs.
partition_by	Optional character vector of columns to partition by (Hive-style). Creates directory structure like `output_dir/SAMPLE_ID=HG00098/data_0.parquet`. Note: DuckLake registration currently requires single Parquet files; when using partition_by, the output_path should point to the partition directory and files should be registered separately.

Details

This is the recommended function for loading VCF data into DuckLake. It uses the `bcf_reader` DuckDB extension for fast VCF→Parquet conversion, which is significantly faster than the nanoarrow streaming path.

Workflow:

1. VCF → Parquet via `vcf_to_parquet_duckdb()` (bcf_reader)
2. Register Parquet in DuckLake catalog

Schema Evolution (`allow_evolution = TRUE`): When loading multiple VCFs with different schemas (e.g., different samples or different annotation fields), enable `allow_evolution` to automatically add new columns to the table schema. This uses DuckLake's `ALTER TABLE ADD COLUMN` which preserves existing data files without rewriting.

Tidy Format (`tidy_format = TRUE`): When building cohort tables from multiple single-sample VCFs, use `tidy_format = TRUE` to get one row per variant-sample combination with a `SAMPLE_ID` column. This format is ideal for downstream analysis and MERGE/UPSERT operations on DuckLake tables.

Partitioning (`partition_by`**):** When using `partition_by`, the output is a Hive-partitioned directory structure. This is useful for large cohorts where you want efficient per-sample queries. DuckDB auto-generates Bloom filters for VARCHAR columns like SAMPLE_ID. Note: For DuckLake, partitioned output requires manual file registration.

Value

Invisibly returns the path to the created Parquet file.

Examples

```
## Not run:
# Build extension
ext_path <- bcf_reader_build(tempdir())

# Setup DuckLake
con <- duckdb::dbConnect(duckdb::duckdb())
ducklake_load(con)
ducklake_attach(con, "catalog.ducklake", "/data/parquet/", alias = "lake")
DBI::dbExecute(con, "USE lake")

# Load first VCF
ducklake_load_vcf(con, "variants", "sample1.vcf.gz", ext_path, threads = 8)

# Load second VCF with different annotations, evolving schema
ducklake_load_vcf(con, "variants", "sample2_vep.vcf.gz", ext_path,
  allow_evolution = TRUE
)

# Load VCF in tidy format (one row per variant-sample)
ducklake_load_vcf(con, "variants_tidy", "cohort.vcf.gz", ext_path,
  tidy_format = TRUE
)

# Query - all columns from both VCFs are available
DBI::dbGetQuery(con, "SELECT CHROM, COUNT(*) FROM variants GROUP BY CHROM")

## End(Not run)
```

---

ducklake_merge *Merge/upsert data into a DuckLake table*

Description

Merge/upsert data into a DuckLake table

Usage

```
ducklake_merge(
  con,
  target,
  source,
  on_cols,
  when_matched = "UPDATE",
  when_not_matched = "INSERT",
  update_cols = NULL
)
```

Arguments

con	DuckDB connection with DuckLake attached.
target	Target table name.
source	Source table/query.
on_cols	Column(s) to match on.
when_matched	Action when matched: "UPDATE", "DELETE", or NULL.
when_not_matched	Action when not matched: "INSERT" or NULL.
update_cols	Columns to update (NULL = all columns).

Value

Number of rows affected.

---

ducklake_options *Get DuckLake configuration options*

Description

Get DuckLake configuration options

Usage

```
ducklake_options(con, catalog = "lake")
```

Arguments

con	DuckDB connection with DuckLake attached.
catalog	DuckLake catalog name.

Value

Data frame with current options.

ducklake_parse_connection_string

Parse DuckLake connection string into components

Description

Parse DuckLake connection string into components

Usage

```
ducklake_parse_connection_string(connection_string)
```

Arguments

connection_string	DuckLake connection string (e.g., "ducklake:path/to/catalog.ducklake").

Value

Named list with components: backend, metadata_path, data_path (if specified).

ducklake_query_snapshot

Query table at a specific snapshot (time travel)

Description

Query table at a specific snapshot (time travel)

Usage

```
ducklake_query_snapshot(con, table, snapshot_id, query = "SELECT * FROM tbl")
```

Arguments

con	DuckDB connection with DuckLake attached.
table	Table name.
snapshot_id	Snapshot version to query.
query	SQL query (use 'tbl' as table alias).

Value

Query result as data frame.

ducklake_register_parquet *Register existing Parquet files in a DuckLake table*

Description

Adds Parquet files that already exist (from prior ETL) to a DuckLake table. This is a catalog-only operation; data files are not copied or moved.

Usage

```
ducklake_register_parquet(
  con,
  table,
  parquet_files,
  create_table = TRUE,
  allow_missing = FALSE,
  ignore_extra_columns = FALSE,
  allow_evolution = FALSE
)
```

Arguments

con	DuckDB connection with DuckLake attached.
table	Target table name (optionally qualified, e.g., "lake.variants").
parquet_files	Character vector of Parquet file paths/URIs.
create_table	Logical, create the table if it doesn't exist. Default: TRUE. When TRUE, schema is inferred from the first Parquet file.
allow_missing	Logical, allow missing columns (filled with defaults). Default: FALSE.
ignore_extra_columns	Logical, ignore extra columns in files. Default: FALSE.
allow_evolution	Logical, evolve table schema by adding new columns from files. Default: FALSE. When TRUE, new columns found in files are added via ALTER TABLE before registration, making all columns queryable.

Details

This function uses DuckLake's `ducklake_add_data_files()` to register external Parquet files in the catalog. The files must already exist and have a schema compatible with the target table.

Schema Evolution (`allow_evolution = TRUE`): When enabled, the function compares each file's schema against the table schema and adds any missing columns via `ALTER TABLE ADD COLUMN` before registration. This allows combining VCF files with different annotations (e.g., VEP columns) into a single table where all columns are queryable.

Value

Invisibly returns the number of files registered.

Examples

```
## Not run:
# Register a Parquet file created by vcf_to_parquet_duckdb()
ducklake_register_parquet(con, "variants", "s3://bucket/variants.parquet")

# Register with schema evolution (add new columns from file)
ducklake_register_parquet(con, "variants", "s3://bucket/vep_variants.parquet",
  allow_evolution = TRUE
)

## End(Not run)
```

---

ducklake_set_commit_message

Set commit message for current transaction

Description

Must be called within a transaction (BEGIN/COMMIT block).

Usage

```
ducklake_set_commit_message(
  con,
  catalog = "lake",
  author,
  message,
  extra_info = NULL
)
```

Arguments

con	DuckDB connection with DuckLake attached.
catalog	DuckLake catalog name.
author	Author name.
message	Commit message.
extra_info	Optional JSON string with extra metadata.

Value

Invisible NULL.

`ducklake_set_option` *Set DuckLake configuration option*

Description

Set DuckLake configuration option

Usage

```
ducklake_set_option(
  con,
  catalog = "lake",
  option,
  value,
  schema = NULL,
  table_name = NULL
)
```

Arguments

`con`	DuckDB connection with DuckLake attached.
`catalog`	DuckLake catalog name.
`option`	Option name (e.g., "parquet_compression", "parquet_row_group_size").
`value`	Option value.
`schema`	Optional schema scope.
`table_name`	Optional table scope.

Details

Common options:

- `parquet_compression`: snappy, zstd, gzip, lz4
- `parquet_row_group_size`: rows per row group (default 122880)
- `target_file_size`: target file size for compaction (default 512MB
- `data_inlining_row_limit`: max rows to inline (default 0)

Value

Invisible NULL.

ducklake_snapshots *List DuckLake snapshots*

Description

List DuckLake snapshots

Usage

```
ducklake_snapshots(con, catalog = "lake")
```

Arguments

con	DuckDB connection with DuckLake attached.
catalog	DuckLake catalog name (alias used in ATTACH).

Value

Data frame with snapshot history.

ducklake_update_secret *Update an existing DuckLake catalog secret*

Description

Update an existing DuckLake catalog secret

Usage

```
ducklake_update_secret(
  con,
  name,
  connection_string,
  data_path = NULL,
  metadata_parameters = list()
)
```

Arguments

con	A DuckDB connection.
name	Secret name to update.
connection_string	New database connection string.
data_path	New default data path. Optional.
metadata_parameters	New named list of metadata parameters.

Value

Invisible NULL.

---

htsfile_path *Get Path to htsfile Executable*

Description

Returns the path to the bundled htsfile executable for identifying file formats.

Usage

```
htsfile_path()
```

Value

A character string containing the path to the htsfile executable.

Examples

```
htsfile_path()
```

---

htslib_bin_dir *Get Path to htslib Binary Directory*

Description

Returns the path to the directory containing htslib executables.

Usage

```
htslib_bin_dir()
```

Details

The directory contains the following tools:

- `annot-tsv` - Annotate TSV files
- `bgzip` - Block gzip compression
- `htsfile` - Identify file format
- `ref-cache` - Reference sequence cache management
- `tabix` - Index and query TAB-delimited files

Value

A character string containing the path to the htslib bin directory.

Examples

```
htslib_bin_dir()
```

---

htslib_capabilities *Get htslib Capabilities*

Description

Returns a named list of all capabilities of the bundled htslib library.

Usage

```
htslib_capabilities()
```

Value

A named list with logical values for each capability:

configure Whether ./configure was used to build.

plugins Whether plugins are enabled.

libcurl Whether libcurl support is enabled.

s3 Whether S3 support is enabled.

gcs Whether Google Cloud Storage support is enabled.

libdeflate Whether libdeflate compression is enabled.

lzma Whether LZMA compression is enabled.

bzip2 Whether bzip2 compression is enabled.

htscodecs Whether htscodecs library is available.

Examples

```
caps <- htslib_capabilities()
caps$libcurl
caps$s3
```

htslib_cflags *Get Compiler Flags for htslib*

Description

Returns the compiler flags (CFLAGS/CPPFLAGS) needed to compile code that uses htslib.

Usage

```
htslib_cflags()
```

Value

A character string containing compiler flags including the `-I` include path.

Examples

```
htslib_cflags()
# Use in Makevars: PKG_CPPFLAGS = $(shell Rscript -e "cat(RBCFTools::htslib_cflags())")
```

htslib_feature_string *Get htslib Feature String*

Description

Returns a human-readable string describing the enabled features in htslib.

Usage

```
htslib_feature_string()
```

Value

A character string describing the enabled features.

Examples

```
htslib_feature_string()
```

htslib_features *Get htslib Features Bitfield*

Description

Returns the raw bitfield of enabled features in htslib.

Usage

```
htslib_features()
```

Value

An integer representing the feature bitfield.

Examples

```
htslib_features()
```

htslib_has_feature *Check for a Specific htslib Feature*

Description

Checks if a specific feature is enabled in the bundled htslib library.

Usage

```
htslib_has_feature(feature_id)

HTS_FEATURE_CONFIGURE

HTS_FEATURE_PLUGINS

HTS_FEATURE_LIBCURL

HTS_FEATURE_S3

HTS_FEATURE_GCS

HTS_FEATURE_LIBDEFLATE

HTS_FEATURE_LZMA

HTS_FEATURE_BZIP2

HTS_FEATURE_HTSCODECS
```

Arguments

`feature_id` An integer feature ID. Use one of the `HTS_FEATURE_*` constants.

Format

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

An object of class `integer` of length 1.

Value

A logical value indicating if the feature is enabled.

Examples

```
# Check for libcurl support (feature ID 1024)
htslib_has_feature(1024L)
```

`htslib_include_dir` *Get htslib Include Directory*

Description

Returns the path to the htslib header files for use in compilation.

Usage

```
htslib_include_dir()
```

Details

This directory contains the htslib headers (e.g., `htslib/hts.h`, `htslib/vcf.h`, etc.). Use this path with `-I` compiler flag when compiling code that uses htslib.

Value

A character string containing the path to the htslib include directory.

Examples

```
htslib_include_dir()
```

`htslib_lib_dir` *Get htslib Library Directory*

Description

Returns the path to the htslib library files for use in linking.

Usage

```
htslib_lib_dir()
```

Details

This directory contains `libhts.a` (static) and `libhts.so` (shared) libraries. Use this path with `-L` linker flag when linking against htslib.

Value

A character string containing the path to the htslib lib directory.

Examples

```
htslib_lib_dir()
```

`htslib_libs` *Get Linker Flags for htslib*

Description

Returns the linker flags needed to link against htslib.

Usage

```
htslib_libs(static = FALSE)
```

Arguments

`static`	Logical. If `TRUE`, returns flags for static linking. If `FALSE` (default), returns flags for dynamic linking.

Details

For dynamic linking, returns `-L<libdir> -lhts`. For static linking, also includes the dependent libraries: `-lpthread -lz -lm -lbz2 -llzma -ldeflate`.

Value

A character string containing linker flags including `-L` library path and `-l` library names.

Examples

```
htslib_libs()
htslib_libs(static = TRUE)
# Use in Makevars: PKG_LIBS = $(shell Rscript -e "cat(RBCFTools::htslib_libs())")
```

---

htslib_plugins_dir *Get Path to htslib Plugins Directory*

Description

Returns the path to the directory containing htslib plugins (e.g., for remote file access via libcurl, S3, GCS).

Usage

```
htslib_plugins_dir()
```

Value

A character string containing the path to the htslib plugins directory.

Examples

```
htslib_plugins_dir()
```

htslib_tools *List Available htslib Tools*

Description

Lists all available tools in the htslib bin directory.

Usage

```
htslib_tools()
```

Value

A character vector of available tool names.

Examples

```
htslib_tools()
```

htslib_version *Get htslib Version*

Description

Returns the version string of the bundled htslib library.

Usage

```
htslib_version()
```

Value

A character string containing the htslib version.

Examples

```
htslib_version()
```

linking_info *Get All Linking Information for RBCFTools*

Description

Returns a list with all paths and flags needed for linking against htslib and bcftools from this package.

Usage

```
linking_info()
```

Value

A named list with the following elements:

htslib_include Path to htslib include directory

htslib_lib Path to htslib library directory

bcftools_lib Path to bcftools library directory

cflags Compiler flags for htslib

htslib_libs Linker flags for htslib (dynamic)

htslib_libs_static Linker flags for htslib (static)

bcftools_libs Linker flags for bcftools

all_libs Combined linker flags for both bcftools and htslib

Examples

```
info <- linking_info()
info$cflags
info$all_libs
```

parquet_kv_metadata *Read Parquet key-value metadata*

Description

Reads the custom key-value metadata stored in a Parquet file's footer. This includes the full VCF header if the file was created with vcf_to_parquet_duckdb with include_metadata = TRUE.

Usage

```
parquet_kv_metadata(file, con = NULL)
```

Arguments

file	Path to Parquet file
con	Optional existing DuckDB connection

Value

A data frame with columns: key, value. Returns empty data frame if no custom metadata exists.

Examples

```
## Not run:
meta <- parquet_kv_metadata("variants.parquet")
# Get the VCF header
vcf_header <- meta[meta$key == "vcf_header", "value"]
cat(vcf_header)

## End(Not run)
```

---

parquet_to_vcf *Convert Parquet back to VCF/BCF format*

Description

Reconstruct a VCF file from Parquet data created by vcf_to_parquet_duckdb. Uses the VCF header stored in Parquet metadata for proper formatting.

Usage

```
parquet_to_vcf(
  input_file,
  output_file,
  header = NULL,
  index = TRUE,
  con = NULL
)
```

Arguments

input_file	Path to input Parquet file (must have VCF metadata)
output_file	Path to output VCF/VCF.GZ/BCF file. Format determined by extension.
header	Optional VCF header string. If NULL (default), reads from Parquet metadata.
index	Logical, if TRUE creates tabix/CSI index for output. Default TRUE.
con	Optional existing DuckDB connection

Value

Invisible path to output file

Examples

```
## Not run:
# Round-trip: VCF -> Parquet -> VCF
vcf_file <- system.file("extdata", "1000G_3samples.vcf.gz", package = "RBCFTools")
ext_path <- bcf_reader_build(tempdir(), verbose = FALSE)

# Convert to Parquet (with metadata)
parquet_file <- tempfile(fileext = ".parquet")
vcf_to_parquet_duckdb(vcf_file, parquet_file, ext_path)

# Convert back to VCF
vcf_out <- tempfile(fileext = ".vcf.gz")
parquet_to_vcf(parquet_file, vcf_out)

## End(Not run)
```

---

print.vcf_duckdb *Print method for vcf_duckdb objects*

Description

Print method for vcf_duckdb objects

Usage

```
## S3 method for class 'vcf_duckdb'
print(x, ...)
```

Arguments

x	A vcf_duckdb object
...	Additional arguments (ignored)

---

print_makevars_config *Print Makevars Configuration for LinkingTo*

Description

Prints example Makevars configuration that can be used by packages that want to link against htslib and/or bcftools via LinkingTo.

Usage

```
print_makevars_config(use_bcftools = FALSE, static = FALSE)
```

Arguments

use_bcftools	Logical. If TRUE, includes bcftools library flags. Default is FALSE (htslib only).
static	Logical. If TRUE, uses static linking flags. Default is FALSE.

Value

Invisibly returns the Makevars text as a character string.

Examples

```
# Print Makevars for htslib only
print_makevars_config()

# Print Makevars for both bcftools and htslib
print_makevars_config(use_bcftools = TRUE)
```

---

ref_cache_path *Get Path to ref-cache Executable*

Description

Returns the path to the bundled ref-cache executable for reference sequence cache management.

Usage

```
ref_cache_path()
```

Value

A character string containing the path to the ref-cache executable.

Examples

```
ref_cache_path()
```

setup_hts_env *Setup Environment for Remote File Access*

Description

Sets the HTS_PATH environment variable to point to the bundled htslib plugins directory. This is required for S3, GCS, and other remote file access via libcurl.

Usage

```
setup_hts_env()
```

Details

Call this function before using bcftools/htslib tools with remote URLs (s3://, gs://, http://, etc.). The function sets HTS_PATH to the package's plugin directory so htslib can find hfile_libcurl.so and hfile_gcs.so.

Value

Invisibly returns the previous value of HTS_PATH (or NA if unset).

Examples

```
setup_hts_env()
# Now bcftools can access S3 URLs
```

tabix_path *Get Path to tabix Executable*

Description

Returns the path to the bundled tabix executable.

Usage

```
tabix_path()
```

Value

A character string containing the path to the tabix executable.

Examples

```
tabix_path()
```

vcf_arrow_schema *Get the Arrow schema for a VCF file*

Description

Reads the header of a VCF/BCF file and returns the corresponding Arrow schema.

Usage

```
vcf_arrow_schema(filename)
```

Arguments

filename	Path to VCF or BCF file

Value

A nanoarrow_schema object

vcf_close_duckdb *Close a VCF DuckDB connection*

Description

Properly closes the DuckDB connection opened by vcf_open_duckdb.

Usage

```
vcf_close_duckdb(vcf, shutdown = TRUE)
```

Arguments

vcf	A vcf_duckdb object returned by vcf_open_duckdb
shutdown	Logical, whether to shutdown the DuckDB instance (default: TRUE)

Value

Invisible NULL

Examples

```
## Not run:
vcf <- vcf_open_duckdb("variants.vcf.gz", ext_path)
# ... do queries ...
vcf_close_duckdb(vcf)

## End(Not run)
```

vcf_count_duckdb *Count variants in a VCF/BCF file*

Description

Fast variant count using DuckDB projection pushdown.

Usage

```
vcf_count_duckdb(
  file,
  extension_path = NULL,
  region = NULL,
  tidy_format = FALSE,
  con = NULL
)
```

Arguments

file	Path to VCF, VCF.GZ, or BCF file
extension_path	Path to the bcf_reader.duckdb_extension file.
region	Optional genomic region for indexed files
tidy_format	Logical, if TRUE counts rows in tidy format (one per variant-sample). Default FALSE returns count of variants.
con	Optional existing DuckDB connection (with extension loaded).

Value

Integer count of variants (or variant-sample combinations if tidy_format=TRUE)

Examples

```
## Not run:
ext_path <- bcf_reader_build(tempdir())
vcf_count_duckdb("variants.vcf.gz", ext_path)
vcf_count_duckdb("variants.vcf.gz", ext_path, region = "chr22")

# Count variant-sample rows (variants * samples)
vcf_count_duckdb("cohort.vcf.gz", ext_path, tidy_format = TRUE)

## End(Not run)
```

vcf_count_per_contig *Get variant counts per contig using bcftools*

Description

Uses bcftools index –stats to get per-contig variant counts. Requires an indexed file.

Usage

```
vcf_count_per_contig(filename)
```

Arguments

filename	Path to VCF/BCF file (must be indexed)

Value

Named integer vector (names = contigs, values = variant counts)

Examples

```
## Not run:
counts <- vcf_count_per_contig("variants.vcf.gz")
# chr1: 12345, chr2: 23456, ...

## End(Not run)
```

vcf_count_variants *Get number of variants using bcftools*

Description

Uses bundled bcftools to count variants efficiently. For indexed files, this is very fast. Can also count per-chromosome.

Usage

```
vcf_count_variants(filename, region = NULL)
```

Arguments

filename	Path to VCF/BCF file
region	Optional region string (e.g., "chr1" or "chr1:1-1000")

Value

Integer count of variants

Examples

```
## Not run:
# Total variants
n <- vcf_count_variants("variants.vcf.gz")

# Variants on chr1
n_chr1 <- vcf_count_variants("variants.vcf.gz", region = "chr1")

## End(Not run)
```

---

vcf_duckdb *DuckDB VCF/BCF Query Utilities*

Description

Functions for querying VCF/BCF files using DuckDB with the bcf_reader extension.

---

vcf_duckdb_connect *Setup DuckDB connection with bcf_reader extension loaded*

Description

Creates a DuckDB connection and loads the bcf_reader extension for VCF/BCF queries.

Usage

```
vcf_duckdb_connect(
  extension_path,
  dbdir = ":memory:",
  read_only = FALSE,
  config = list()
)
```

Arguments

extension_path	Path to the bcf_reader.duckdb_extension file. Must be explicitly provided.
dbdir	Database directory. Default ":memory:" for in-memory database.
read_only	Logical, whether to open in read-only mode. Default FALSE.
config	Named list of DuckDB configuration options.

Value

A DuckDB connection object with bcf_reader extension loaded

Examples

```
## Not run:
# First build the extension
ext_path <- bcf_reader_build(tempdir())

# Then connect
con <- vcf_duckdb_connect(ext_path)
DBI::dbGetQuery(con, "SELECT * FROM bcf_read('variants.vcf.gz') LIMIT 10")
DBI::dbDisconnect(con)

## End(Not run)
```

---

vcf_get_contig_lengths

Get contig lengths from VCF/BCF file

Description

Extracts contig names and lengths from the VCF/BCF header.

Usage

```
vcf_get_contig_lengths(filename)
```

Arguments

filename	Path to VCF/BCF file

Value

Named integer vector (names = contigs, values = lengths)

Examples

```
## Not run:
lengths <- vcf_get_contig_lengths("variants.vcf.gz")

## End(Not run)
```

vcf_get_contigs *Get contig names from VCF/BCF file*

Description

Extracts contig names from the VCF/BCF header using htslib.

Usage

```
vcf_get_contigs(filename)
```

Arguments

filename	Path to VCF/BCF file

Value

Character vector of contig names

Examples

```
## Not run:
contigs <- vcf_get_contigs("variants.vcf.gz")

## End(Not run)
```

vcf_has_index *Check if VCF/BCF file has an index*

Description

Uses htslib to robustly check for index presence. Works with local files, remote URLs (S3, GCS, HTTP), and custom index paths.

Usage

```
vcf_has_index(filename, index = NULL)
```

Arguments

filename	Path to VCF/BCF file
index	Optional explicit index path

Value

Logical indicating if index exists

Examples

```
## Not run:
vcf_has_index("variants.vcf.gz")
vcf_has_index("s3://bucket/file.vcf.gz")
vcf_has_index("file.vcf.gz", index = "custom.tbi")

## End(Not run)
```

`vcf_header_metadata` *Extract VCF header for Parquet key-value storage*

Description

Extracts the full VCF header from a file for embedding in Parquet metadata. This allows round-tripping back to VCF format by preserving all header information (INFO, FORMAT, FILTER definitions, contigs, samples).

Usage

```
vcf_header_metadata(file)
```

Arguments

`file`	Path to VCF, VCF.GZ, or BCF file

Value

A named list with two elements:

- `vcf_header`: The complete VCF header (all lines starting with #)
- `RBCFTools_version`: Package version that created the Parquet

Examples

```
## Not run:
vcf_file <- system.file("extdata", "1000G_3samples.vcf.gz", package = "RBCFTools")
meta <- vcf_header_metadata(vcf_file)
cat(meta$vcf_header)

## End(Not run)
```

vcf_open_arrow *Create an Arrow stream from a VCF/BCF file*

Description

Opens a VCF or BCF file and creates an Arrow array stream that produces record batches. This enables efficient, streaming access to variant data in Arrow format.

Usage

```
vcf_open_arrow(
  filename,
  batch_size = 10000L,
  region = NULL,
  samples = NULL,
  include_info = TRUE,
  include_format = TRUE,
  index = NULL,
  threads = 0L,
  parse_vep = FALSE,
  vep_tag = NULL,
  vep_columns = NULL,
  vep_transcript = c("first", "all")
)
```

Arguments

filename	Path to VCF or BCF file
batch_size	Number of records per batch (default: 10000)
region	Optional region string for filtering (e.g., "chr1:1000-2000")
samples	Optional sample filter (comma-separated names or "-" prefixed to exclude)
include_info	Include INFO fields in output (default: TRUE)
include_format	Include FORMAT/sample data in output (default: TRUE)
index	Optional index file path. If NULL (default), uses auto-detection: VCF files try .tbi first, then .csi; BCF files use .csi only. Useful for non-standard index locations or presigned URLs with different paths. Alternatively, use htslib ##idx## syntax in filename (e.g., "file.vcf.gz##idx##custom.tbi"). Note: Index is only required for region queries; whole-file streaming needs no index.
threads	Number of decompression threads (default: 0 = auto)
parse_vep	Enable VEP/BCSQ/ANN annotation parsing (default: FALSE). When TRUE, annotation fields are parsed and added as typed columns.
vep_tag	Annotation tag to parse ("CSQ", "BCSQ", "ANN") or NULL for auto-detect.
vep_columns	Character vector of VEP fields to extract, or NULL for all fields.
vep_transcript	Which transcript to extract: "first" (default) or "all". "first" returns scalar columns (one value per variant). "all" returns list columns (all transcripts per variant).

Value

A nanoarrow_array_stream object

Examples

```
## Not run:
# Basic usage
stream <- vcf_open_arrow("variants.vcf.gz")

# Read batches
while (!is.null(batch <- stream$get_next())) {
  # Process batch...
  print(nanoarrow::convert_array(batch))
}

# With region filter
stream <- vcf_open_arrow("variants.vcf.gz", region = "chr1:1-1000000")

# With custom index file (useful for presigned URLs or non-standard locations)
stream <- vcf_open_arrow("variants.vcf.gz", index = "custom_path.tbi", region = "chr1")

# Convert to data frame
df <- vcf_to_arrow("variants.vcf.gz", as = "data.frame")

# Write to parquet (uses DuckDB, no arrow package needed)
vcf_to_parquet_arrow("variants.vcf.gz", "variants.parquet")

## End(Not run)
```

---

vcf_open_duckdb *Open a VCF/BCF file as a DuckDB table or view*

Description

Creates a DuckDB connection with the VCF data loaded as a table or view. Supports in-memory or file-backed databases, tidy format output, parallel loading by chromosome, column selection, and optional Hive partitioning.

Usage

```
vcf_open_duckdb(
  file,
  extension_path,
  table_name = "variants",
  as_view = TRUE,
  dbdir = ":memory:",
  columns = NULL,
  region = NULL,
```

```
  tidy_format = FALSE,
  threads = 1L,
  partition_by = NULL,
  overwrite = FALSE,
  config = list()
)
```

Arguments

file	Path to VCF, VCF.GZ, or BCF file
extension_path	Path to the bcf_reader.duckdb_extension file.
table_name	Name for the table/view (default: "variants")
as_view	Logical, create a VIEW instead of materializing a TABLE (default: TRUE). Views are instant to create but queries re-read the VCF each time. Tables are slower to create but subsequent queries are fast.
dbdir	Database directory. Default ":memory:" for in-memory database. Use a file path for persistent storage (e.g., "variants.duckdb").
columns	Optional character vector of columns to include. NULL for all.
region	Optional genomic region filter (e.g., "chr1:1000-2000"). Requires an indexed VCF.
tidy_format	Logical, if TRUE loads data in tidy (long) format with one row per variant-sample combination and a SAMPLE_ID column. Default FALSE.
threads	Number of threads for parallel loading (default: 1). When > 1 and VCF is indexed: • For views (as_view = TRUE): Creates a UNION ALL view of per-contig bcf_read() calls. DuckDB parallelizes execution at query time. • For tables (as_view = FALSE): Loads each chromosome in parallel then unions into a single table.
partition_by	Optional character vector of columns to partition by when creating a table (ignored for views). Creates a partitioned table for efficient filtering. Only supported for file-backed databases.
overwrite	Logical, drop existing table/view if it exists (default: FALSE).
config	Named list of DuckDB configuration options.

Value

A list with:

con	DuckDB connection with extension loaded
table	Name of the created table/view
is_view	Logical indicating if a view was created
file	Path to the source VCF file
dbdir	Database directory
tidy_format	Whether tidy format was used
row_count	Number of rows (NULL for views)

Examples

```
## Not run:
ext_path <- bcf_reader_build(tempdir())

# Open as lazy view (default - instant creation, re-reads VCF each query)
vcf <- vcf_open_duckdb("variants.vcf.gz", ext_path)
DBI::dbGetQuery(vcf$con, "SELECT * FROM variants WHERE CHROM = '22'")
vcf_close_duckdb(vcf)

# Parallel view (UNION ALL of per-contig reads, parallelized at query time)
vcf <- vcf_open_duckdb("wgs.vcf.gz", ext_path, threads = 8)

# Open as materialized table (slower to create, fast repeated queries)
vcf <- vcf_open_duckdb("variants.vcf.gz", ext_path, as_view = FALSE)
DBI::dbGetQuery(vcf$con, "SELECT COUNT(*) FROM variants")

# Tidy format with specific columns
vcf <- vcf_open_duckdb("cohort.vcf.gz", ext_path,
  tidy_format = TRUE,
  columns = c("CHROM", "POS", "REF", "ALT", "SAMPLE_ID", "FORMAT_GT")
)

# Parallel table loading for large files
vcf <- vcf_open_duckdb("wgs.vcf.gz", ext_path, as_view = FALSE, threads = 8)

# Persistent file-backed database
vcf <- vcf_open_duckdb("variants.vcf.gz", ext_path,
  dbdir = "my_variants.duckdb"
)

# Partitioned table for efficient sample queries
vcf <- vcf_open_duckdb("cohort.vcf.gz", ext_path,
  dbdir = "cohort.duckdb",
  tidy_format = TRUE,
  partition_by = "SAMPLE_ID"
)

## End(Not run)
```

---

vcf_query_arrow *Query VCF/BCF with DuckDB*

Description

Enables SQL queries on VCF files using DuckDB. This allows powerful filtering, aggregation, and joining operations.

Usage

```
vcf_query_arrow(vcf_files, query, ...)
```

Arguments

vcf_files	Character vector of VCF file paths
query	SQL query string. Use "vcf" as the table name.
...	Additional arguments passed to vcf_open_arrow

Value

Query result as a data frame

Examples

```
## Not run:
# Count variants per chromosome
vcf_query_arrow(
  "variants.vcf.gz",
  "SELECT CHROM, COUNT(*) as n FROM vcf GROUP BY CHROM"
)

# Filter high-quality variants
vcf_query_arrow(
  "variants.vcf.gz",
  "SELECT * FROM vcf WHERE QUAL > 30"
)

# Join multiple VCF files
vcf_query_arrow(
  c("sample1.vcf.gz", "sample2.vcf.gz"),
  "SELECT * FROM vcf WHERE POS BETWEEN 1000 AND 2000"
)

## End(Not run)
```

---

vcf_query_duckdb *Query a VCF/BCF file using DuckDB SQL*

Description

Execute a SQL query against a VCF/BCF file using the bcf_reader extension. The file is exposed as a table via the `bcf_read()` function.

Usage

```
vcf_query_duckdb(
  file,
  extension_path = NULL,
  query = NULL,
  region = NULL,
```

```
  tidy_format = FALSE,
  con = NULL
)
```

Arguments

file	Path to VCF, VCF.GZ, or BCF file
extension_path	Path to the bcf_reader.duckdb_extension file.
query	SQL query string. Use `bcf_read('{file}')` to reference the file, or if NULL, returns all rows with `SELECT * FROM bcf_read('{file}')`.
region	Optional genomic region for indexed files (e.g., "chr1:1000-2000")
tidy_format	Logical, if TRUE returns data in tidy (long) format with one row per variant-sample combination and a SAMPLE_ID column. Default FALSE.
con	Optional existing DuckDB connection (with extension already loaded). If provided, extension_path is ignored.

Value

A data.frame with query results

Examples

```
## Not run:
# First build the extension
ext_path <- bcf_reader_build(tempdir())

# Basic query - get all variants
vcf_query_duckdb("variants.vcf.gz", ext_path)

# Count variants
vcf_query_duckdb("variants.vcf.gz", ext_path,
  query = "SELECT COUNT(*) FROM bcf_read('{file}')"
)

# Filter by chromosome
vcf_query_duckdb("variants.vcf.gz", ext_path,
  query = "SELECT CHROM, POS, REF, ALT FROM bcf_read('{file}') WHERE CHROM = '22'"
)

# Region query (requires index)
vcf_query_duckdb("variants.vcf.gz", ext_path, region = "chr1:1000000-2000000")

# Tidy format - one row per variant-sample
vcf_query_duckdb("cohort.vcf.gz", ext_path, tidy_format = TRUE)

# Reuse connection for multiple queries
con <- vcf_duckdb_connect(ext_path)
vcf_query_duckdb("file1.vcf.gz", con = con)
vcf_query_duckdb("file2.vcf.gz", con = con)
DBI::dbDisconnect(con, shutdown = TRUE)
```

```
## End(Not run)
```

`vcf_read_vep` *Read VCF with parsed VEP annotations*

Description

Opens a VCF file and parses VEP/BCSQ/ANN annotations into structured columns. This is a convenience wrapper around `vcf_open_arrow` with VEP parsing enabled.

Usage

```
vcf_read_vep(filename, vep_tag = NULL, vep_columns = NULL, ...)
```

Arguments

`filename`	Path to VCF/BCF file
`vep_tag`	Annotation tag to parse ("CSQ", "BCSQ", "ANN") or NULL for auto-detection
`vep_columns`	Character vector of VEP fields to extract, or NULL for all fields
`...`	Additional arguments passed to `vcf_to_arrow`

Value

Data frame with VCF columns plus parsed VEP fields as separate columns prefixed with the tag name (e.g., "CSQ_Consequence", "CSQ_SYMBOL", etc.)

Examples

```
## Not run:
df <- vcf_read_vep("annotated.vcf.gz",
  vep_columns = c("Consequence", "SYMBOL", "AF", "gnomAD_AF")
)

# Filter by gnomAD frequency
rare <- df[!is.na(df$CSQ_gnomAD_AF) & df$CSQ_gnomAD_AF < 0.001, ]

## End(Not run)
```

vcf_samples_duckdb *List samples in a VCF/BCF file using DuckDB*

Description

Extract sample names from FORMAT column names.

Usage

```
vcf_samples_duckdb(file, extension_path = NULL, con = NULL)
```

Arguments

file	Path to VCF, VCF.GZ, or BCF file
extension_path	Path to the bcf_reader.duckdb_extension file.
con	Optional existing DuckDB connection (with extension loaded).

Value

Character vector of sample names

Examples

```
## Not run:
ext_path <- bcf_reader_build(tempdir())
vcf_samples_duckdb("variants.vcf.gz", ext_path)

## End(Not run)
```

vcf_schema_duckdb *Get VCF/BCF schema using DuckDB*

Description

Returns the column names and types for a VCF/BCF file as seen by DuckDB.

Usage

```
vcf_schema_duckdb(file, extension_path = NULL, tidy_format = FALSE, con = NULL)
```

Arguments

file	Path to VCF, VCF.GZ, or BCF file
extension_path	Path to the bcf_reader.duckdb_extension file.
tidy_format	Logical, if TRUE returns schema for tidy format. Default FALSE.
con	Optional existing DuckDB connection (with extension loaded).

Value

A data.frame with column_name and column_type

Examples

```
## Not run:
ext_path <- bcf_reader_build(tempdir())
vcf_schema_duckdb("variants.vcf.gz", ext_path)

# Compare wide vs tidy schemas
vcf_schema_duckdb("cohort.vcf.gz", ext_path) # FORMAT_GT_Sample1, FORMAT_GT_Sample2...
vcf_schema_duckdb("cohort.vcf.gz", ext_path, tidy_format = TRUE) # SAMPLE_ID, FORMAT_GT

## End(Not run)
```

---

vcf_summary_duckdb *Summary statistics for a VCF/BCF file using DuckDB*

Description

Get summary statistics including variant counts per chromosome.

Usage

```
vcf_summary_duckdb(file, extension_path = NULL, con = NULL)
```

Arguments

file	Path to VCF, VCF.GZ, or BCF file
extension_path	Path to the bcf_reader.duckdb_extension file.
con	Optional existing DuckDB connection (with extension loaded).

Value

A list with total_variants, n_samples, and variants_per_chrom

Examples

```
## Not run:
ext_path <- bcf_reader_build(tempdir())
vcf_summary_duckdb("variants.vcf.gz", ext_path)

## End(Not run)
```

vcf_to_arrow *Read VCF/BCF file into a data frame or list of batches*

Description

Convenience function to read an entire VCF file into memory as an R data structure.

Usage

```
vcf_to_arrow(filename, as = c("tibble", "data.frame", "batches"), ...)
```

Arguments

filename	Path to VCF or BCF file
as	Character string specifying output format: "tibble", "data.frame", or "batches" (list of nanoarrow arrays)
...	Additional arguments passed to vcf_open_arrow

Value

Depends on as parameter:

- "tibble": A tibble
- "data.frame": A data.frame
- "batches": A list of nanoarrow_array objects

vcf_to_arrow_ipc *Write VCF/BCF to Arrow IPC format*

Description

Converts a VCF/BCF file to Arrow IPC stream format for efficient storage and interoperability with Arrow-compatible tools. Uses nanoarrow's native IPC writer for streaming output.

Usage

```
vcf_to_arrow_ipc(input_vcf, output_ipc, ...)
```

Arguments

input_vcf	Path to input VCF or BCF file
output_ipc	Path for output Arrow IPC file (typically .arrows extension)
...	Additional arguments passed to vcf_open_arrow

Value

Invisibly returns the output path

Examples

```
## Not run:
vcf_to_arrow_ipc("variants.vcf.gz", "variants.arrows")

# Read back with nanoarrow
stream <- nanoarrow::read_nanoarrow("variants.arrows")
df <- as.data.frame(stream)

# Or query with DuckDB
library(duckdb)
con <- dbConnect(duckdb())
dbGetQuery(con, "SELECT * FROM 'variants.arrows' LIMIT 10")

## End(Not run)
```

---

vcf_to_parquet_arrow *Write VCF/BCF to Parquet format*

Description

Converts a VCF/BCF file to Apache Parquet format for efficient storage and querying with tools like DuckDB, Spark, or Python pandas/polars.

Usage

```
vcf_to_parquet_arrow(
  input_vcf,
  output_parquet,
  compression = "zstd",
  row_group_size = 100000L,
  streaming = FALSE,
  threads = 1L,
  index = NULL,
  ...
)
```

Arguments

input_vcf	Path to input VCF or BCF file
output_parquet	Path for output Parquet file
compression	Compression codec: "snappy", "gzip", "zstd", "lz4", "uncompressed"
row_group_size	Number of rows per row group (default: 100000)

streaming	Use streaming mode for large files. When TRUE, writes to a temporary Arrow IPC file first (via nanoarrow), then converts to Parquet via DuckDB. This avoids loading the entire VCF into R memory. Requires the DuckDB nanoarrow community extension. Default is FALSE.
threads	Number of parallel threads for processing (default: 1). When threads > 1 and file is indexed, uses parallel processing by splitting work across chromosomes/contigs. Each thread processes different regions simultaneously. Requires indexed file. See vcf_to_parquet_parallel_arrow for details.
index	Optional explicit index file path
...	Additional arguments passed to vcf_open_arrow

Details

Processing Modes:

1. **Standard mode** (`streaming = FALSE, threads = 1`): Loads entire VCF into memory as data.frame before writing. Fast for small-medium files.
2. **Streaming mode** (`streaming = TRUE, threads = 1`): Two-stage streaming via temporary Arrow IPC file. Minimal memory usage for large files.
3. **Parallel mode** (`threads > 1`): Requires indexed file. Splits work by chromosomes, processing multiple regions simultaneously. Near-linear speedup with thread count. Best for whole-genome VCFs.

Value

Invisibly returns the output path

Examples

```
## Not run:
# Standard mode (fast, loads into memory)
vcf_to_parquet_arrow("variants.vcf.gz", "variants.parquet")

# Streaming mode for large files (low memory)
vcf_to_parquet_arrow("huge.vcf.gz", "huge.parquet", streaming = TRUE)

# Parallel mode for whole-genome VCF (requires index)
vcf_to_parquet_arrow("wgs.vcf.gz", "wgs.parquet", threads = 8)

# Parallel + streaming for massive files
vcf_to_parquet_arrow("wgs.vcf.gz", "wgs.parquet", threads = 16, streaming = TRUE)

# With zstd compression
vcf_to_parquet_arrow("variants.vcf.gz", "variants.parquet", compression = "zstd")

# Query with DuckDB
library(duckdb)
con <- dbConnect(duckdb())
dbGetQuery(con, "SELECT CHROM, POS, REF FROM 'variants.parquet' WHERE CHROM = 'chr1'")
```

```
## End(Not run)
```

`vcf_to_parquet_duckdb` *Export VCF/BCF to Parquet using DuckDB*

Description

Convert a VCF/BCF file to Parquet format for fast subsequent queries.

Usage

```
vcf_to_parquet_duckdb(
  input_file,
  output_file,
  extension_path = NULL,
  columns = NULL,
  region = NULL,
  compression = "zstd",
  row_group_size = 100000L,
  threads = 1L,
  tidy_format = FALSE,
  partition_by = NULL,
  include_metadata = TRUE,
  con = NULL
)
```

Arguments

`input_file`	Path to input VCF, VCF.GZ, or BCF file
`output_file`	Path to output Parquet file or directory (when using partition_by)
`extension_path`	Path to the bcf_reader.duckdb_extension file.
`columns`	Optional character vector of columns to include. NULL for all.
`region`	Optional genomic region to export (requires index)
`compression`	Parquet compression: "snappy", "zstd", "gzip", or "none"
`row_group_size`	Number of rows per row group (default: 100000)
`threads`	Number of parallel threads for processing (default: 1). When threads > 1 and file is indexed, uses parallel processing by splitting work across chromosomes/contigs. See `vcf_to_parquet_duckdb_parallel`.
`tidy_format`	Logical, if TRUE exports data in tidy (long) format with one row per variant-sample combination and a SAMPLE_ID column. Default FALSE.
`partition_by`	Optional character vector of columns to partition by (Hive-style). Creates a directory structure like `output_dir/SAMPLE_ID=HG00098/data_0.parquet`. Particularly useful with `tidy_format = TRUE` to partition by SAMPLE_ID for efficient per-sample queries. DuckDB auto-generates Bloom filters for VARCHAR columns like SAMPLE_ID, enabling fast row group pruning.

include_metadata
: Logical, if TRUE embeds the full VCF header as Parquet key-value metadata. Default TRUE. This preserves all VCF schema information (INFO, FORMAT, FILTER definitions, contigs, samples) enabling round-trip back to VCF format. Use parquet_kv_metadata to read the header back. Note: Not supported with partition_by (Parquet limitation for partitioned writes).

con
: Optional existing DuckDB connection (with extension loaded).

Value

Invisible path to output file/directory

Examples

```
## Not run:
ext_path <- bcf_reader_build(tempdir())

# Export entire file with metadata
vcf_to_parquet_duckdb("variants.vcf.gz", "variants.parquet", ext_path)

# Read back the embedded metadata
parquet_kv_metadata("variants.parquet")

# Export specific columns
vcf_to_parquet_duckdb("variants.vcf.gz", "variants_slim.parquet", ext_path,
  columns = c("CHROM", "POS", "REF", "ALT", "INFO_AF")
)

# Export a region
vcf_to_parquet_duckdb("variants.vcf.gz", "chr22.parquet", ext_path,
  region = "chr22"
)

# Export in tidy format (one row per variant-sample)
vcf_to_parquet_duckdb("cohort.vcf.gz", "cohort_tidy.parquet", ext_path,
  tidy_format = TRUE
)

# Tidy format with Hive partitioning by SAMPLE_ID (efficient per-sample queries)
vcf_to_parquet_duckdb("cohort.vcf.gz", "cohort_partitioned/", ext_path,
  tidy_format = TRUE,
  partition_by = "SAMPLE_ID"
)

# Partition by both CHROM and SAMPLE_ID for large cohorts
vcf_to_parquet_duckdb("wgs_cohort.vcf.gz", "wgs_partitioned/", ext_path,
  tidy_format = TRUE,
  partition_by = c("CHROM", "SAMPLE_ID")
)

# Parallel mode for whole-genome VCF (requires index)
vcf_to_parquet_duckdb("wgs.vcf.gz", "wgs.parquet", ext_path, threads = 8)
```

```
## End(Not run)
```

`vcf_to_parquet_duckdb_parallel` *Parallel VCF to Parquet conversion using DuckDB*

Description

Processes VCF/BCF file in parallel by splitting work across chromosomes/contigs using the DuckDB bcf_reader extension. Requires an indexed file. Each thread processes a different chromosome, then results are merged into a single Parquet file.

Usage

```
vcf_to_parquet_duckdb_parallel(
  input_file,
  output_file,
  extension_path = NULL,
  threads = parallel::detectCores(),
  compression = "zstd",
  row_group_size = 100000L,
  columns = NULL,
  tidy_format = FALSE,
  partition_by = NULL,
  con = NULL
)
```

Arguments

`input_file`	Path to input VCF/BCF file (must be indexed)
`output_file`	Path for output Parquet file
`extension_path`	Path to the bcf_reader.duckdb_extension file.
`threads`	Number of parallel threads (default: auto-detect)
`compression`	Parquet compression codec
`row_group_size`	Row group size
`columns`	Optional character vector of columns to include
`tidy_format`	Logical, if TRUE exports data in tidy (long) format. Default FALSE.
`partition_by`	Optional character vector of columns to partition by (Hive-style). Creates directory structure like `output_dir/SAMPLE_ID=HG00098/data_0.parquet`. Note: When using partition_by, each contig's data is partitioned separately then merged into the final partitioned output.
`con`	Optional existing DuckDB connection (with extension loaded).

Details

This function:

1. Checks for index (required for parallel processing)
2. Extracts contig names from header
3. Processes each contig in parallel using multiple R processes
4. Writes each contig to a temporary Parquet file
5. Merges all temporary files into final output using DuckDB

Contigs that return no variants are skipped automatically.

When `partition_by` is specified, the function creates a Hive-partitioned directory structure. This is especially useful with `tidy_format = TRUE` and `partition_by = "SAMPLE_ID"` for efficient per-sample queries on large cohorts. DuckDB auto-generates Bloom filters for VARCHAR columns like SAMPLE_ID.

Value

Invisibly returns the output path

See Also

`vcf_to_parquet_duckdb` for single-threaded conversion

Examples

```
## Not run:
ext_path <- bcf_reader_build(tempdir())

# Use 8 threads
vcf_to_parquet_duckdb_parallel("wgs.vcf.gz", "wgs.parquet", ext_path, threads = 8)

# With specific columns
vcf_to_parquet_duckdb_parallel(
  "wgs.vcf.gz", "wgs.parquet", ext_path,
  threads = 16,
  columns = c("CHROM", "POS", "REF", "ALT")
)

# Tidy format output
vcf_to_parquet_duckdb_parallel("wgs.vcf.gz", "wgs_tidy.parquet", ext_path,
  threads = 8, tidy_format = TRUE
)

# Tidy format with Hive partitioning by SAMPLE_ID
vcf_to_parquet_duckdb_parallel("wgs_cohort.vcf.gz", "wgs_partitioned/", ext_path,
  threads = 8, tidy_format = TRUE, partition_by = "SAMPLE_ID"
)

## End(Not run)
```

`vcf_to_parquet_parallel_arrow` *Parallel VCF to Parquet conversion*

Description

Processes VCF/BCF file in parallel by splitting work across chromosomes/contigs. Requires an indexed file. Each thread processes a different chromosome, then results are merged into a single Parquet file.

Usage

```
vcf_to_parquet_parallel_arrow(
  input_vcf,
  output_parquet,
  threads = parallel::detectCores(),
  compression = "zstd",
  row_group_size = 100000L,
  streaming = FALSE,
  index = NULL,
  ...
)
```

Arguments

`input_vcf`	Path to input VCF/BCF file (must be indexed)
`output_parquet`	Path for output Parquet file
`threads`	Number of parallel threads (default: auto-detect)
`compression`	Compression codec
`row_group_size`	Row group size
`streaming`	Use streaming mode
`index`	Optional explicit index path
`...`	Additional arguments passed to vcf_open_arrow

Details

This function:

1. Checks for index (required for parallel processing)
2. Extracts contig names from header
3. Processes each contig in parallel using multiple R processes
4. Writes each contig to a temporary Parquet file
5. Merges all temporary files into final output using DuckDB

Contigs that return no variants are skipped automatically.

Value

Invisibly returns the output path

Examples

```
## Not run:
# Use 8 threads
vcf_to_parquet_parallel_arrow("wgs.vcf.gz", "wgs.parquet", threads = 8)

# With streaming mode for large files
vcf_to_parquet_parallel_arrow(
    "huge.vcf.gz", "huge.parquet",
    threads = 16, streaming = TRUE
)

## End(Not run)
```

`vep_detect_tag` *Detect VEP annotation tag in VCF file*

Description

Checks for the presence of CSQ, BCSQ, or ANN annotation tags in the VCF header and returns the first one found.

Usage

```
vep_detect_tag(filename)
```

Arguments

`filename`	Path to VCF/BCF file

Value

Character string with tag name ("CSQ", "BCSQ", or "ANN"), or NA if no annotation found

Examples

```
## Not run:
vep_detect_tag("annotated.vcf.gz")  # Returns "CSQ"

## End(Not run)
```

vep_get_schema *Get VEP annotation schema from VCF header*

Description

Parses the VEP/BCSQ/ANN header to extract field names and inferred types. Types are inferred using bcftools split-vep conventions.

Usage

```
vep_get_schema(filename, tag = NULL)
```

Arguments

filename	Path to VCF/BCF file
tag	Optional annotation tag ("CSQ", "BCSQ", "ANN"). If NULL (default), auto-detects.

Value

Data frame with columns:

name Field name (e.g., "Consequence", "SYMBOL", "AF")

type Inferred type ("Integer", "Float", "String")

index Position in pipe-delimited string (0-based)

is_list Whether field can have multiple values

The tag name is stored as an attribute.

Examples

```
## Not run:
schema <- vep_get_schema("vep_annotated.vcf.gz")
print(schema)
#       name    type index is_list
# 1   Allele  String     0   FALSE
# 2   Consequence String  1    TRUE
# 3   IMPACT  String     2   FALSE
# ...
attr(schema, "tag")  # "CSQ"

## End(Not run)
```

vep_has_annotation *Check if VCF has VEP-style annotations*

Description

Check if VCF has VEP-style annotations

Usage

```
vep_has_annotation(filename)
```

Arguments

filename	Path to VCF/BCF file

Value

Logical indicating presence of CSQ, BCSQ, or ANN

Examples

```
## Not run:
if (vep_has_annotation("file.vcf.gz")) {
  schema <- vep_get_schema("file.vcf.gz")
}

## End(Not run)
```

vep_infer_type *Infer type from VEP field name*

Description

Uses bcftools split-vep conventions to infer the type of a VEP field from its name.

Usage

```
vep_infer_type(field_name)
```

Arguments

field_name	Character vector of field names

Details

Known integer fields: DISTANCE, STRAND, TSL, GENE_PHENO, HGVS_OFFSET, MOTIF_POS, existing_*ORFs, SpliceAI_pred_DP_*

Known float fields: AF, AF (e.g., gnomAD_AF), MAX_AF, MOTIF_SCORE_CHANGE, SpliceAI_pred_DS_*

All others default to String.

Value

Character vector of inferred types ("Integer", "Float", "String")

Examples

```
vep_infer_type(c("SYMBOL", "AF", "gnomAD_AF", "DISTANCE", "SpliceAI_pred_DS_AG"))
# [1] "String" "Float" "Float" "Integer" "Float"
```

---

vep_list_fields *List VEP annotation fields in a VCF file*

Description

Convenience function to display available VEP fields and their types.

Usage

```
vep_list_fields(filename)
```

Arguments

filename	Path to VCF/BCF file

Value

Invisibly returns the schema data frame

Examples

```
## Not run:
vep_list_fields("annotated.vcf.gz")
# VEP Annotation Tag: CSQ
# Fields (78 total):
#   1. Allele (String)
#   2. Consequence (String, list)
#   3. IMPACT (String)
#   ...

## End(Not run)
```

vep_parse_record *Parse VEP annotation string*

Description

Parses a CSQ/BCSQ/ANN annotation string into a structured list of data frames, one per transcript/consequence.

Usage

```
vep_parse_record(csq_value, filename, schema = NULL)
```

Arguments

csq_value	Raw annotation string (pipe-delimited, comma-separated for multiple transcripts)
filename	Path to VCF file (for schema extraction)
schema	Optional pre-parsed schema from vep_get_schema(). If NULL, extracted from filename.

Value

List of data frames, one per transcript. Each data frame has one row with columns corresponding to annotation fields, properly typed.

Examples

```
## Not run:
# Get a CSQ value from a VCF
csq <- "A|missense_variant|MODERATE|BRCA1|..."
result <- vep_parse_record(csq, "annotated.vcf.gz")
result[[1]]$Consequence  # "missense_variant"
result[[1]]$AF           # 0.001 (numeric)

## End(Not run)
```

Index