

# Package: arkdb (via r-universe)

November 27, 2024

**Version** 0.0.18

**Title** Archive and Unarchive Databases Using Flat Files

**Description** Flat text files provide a robust, compressible, and portable way to store tables from databases. This package provides convenient functions for exporting tables from relational database connections into compressed text files and streaming those text files back into a database without requiring the whole table to fit in working memory.

**URL** <https://github.com/ropensci/arkdb>,  
<https://docs.ropensci.org/arkdb/>

**BugReports** <https://github.com/ropensci/arkdb/issues>

**License** MIT + file LICENSE

**Encoding** UTF-8

**ByteCompile** true

**VignetteBuilder** knitr

**RoxygenNote** 7.2.1

**Roxygen** list(markdown = TRUE)

**Depends** R (>= 4.0)

**Imports** DBI, tools, utils

**Suggests** arrow, R.utils, progress, readr, spelling, dplyr, dbplyr, nycflights13, testthat, knitr, covr, fs, rmarkdown, RSQLite, duckdb, vroom, utf8, future.apply

**Language** en-US

**Repository** <https://ropensci.r-universe.dev>

**RemoteUrl** <https://github.com/ropensci/arkdb>

**RemoteRef** master

**RemoteSha** 0a836d03eac94c3d7d9f697d190546df2b3bd3d0

## Contents

arkdb-package . . . . .	2
ark . . . . .	3
arkdb_delete_db . . . . .	5
local_db . . . . .	5
local_db_disconnect . . . . .	7
process_chunks . . . . .	8
streamable_base_csv . . . . .	8
streamable_base_tsv . . . . .	9
streamable_parquet . . . . .	10
streamable_readr_csv . . . . .	10
streamable_readr_tsv . . . . .	11
streamable_table . . . . .	11
streamable_vroom . . . . .	12
unark . . . . .	12

<b>Index</b>	<b>15</b>
--------------	-----------

---

arkdb-package	<i>arkdb: Archive and Unarchive Databases Using Flat Files</i>
---------------	--

---

## Description

Flat text files provide a more robust, compressible, and portable way to store tables. This package provides convenient functions for exporting tables from relational database connections into compressed text files and streaming those text files back into a database without requiring the whole table to fit in working memory.

## Details

It has two functions:

- `ark()`: archive a database into flat files, chunk by chunk.
- `unark()`: Unarchive flat files back into a database connection.

arkdb will work with any DBI supported connection. This makes it a convenient and robust way to migrate between different databases as well.

## Author(s)

**Maintainer:** Carl Boettiger <cboettig@gmail.com> ([ORCID](#)) [copyright holder]

Other contributors:

- Richard FitzJohn [contributor]
- Brandon Bertelsen <brandon@bertelsen.ca> [contributor]

**See Also**

Useful links:

- <https://github.com/ropensci/arkdb>
- Report bugs at <https://github.com/ropensci/arkdb/issues>

---

ark

*Archive tables from a database as flat files*


---

**Description**

Archive tables from a database as flat files

**Usage**

```
ark(
  db_con,
  dir,
  streamable_table = streamable_base_tsv(),
  lines = 50000L,
  compress = c("bzip2", "gzip", "xz", "none"),
  tables = list_tables(db_con),
  method = c("keep-open", "window", "window-parallel", "sql-window"),
  overwrite = "ask",
  filter_statement = NULL,
  filenames = NULL,
  callback = NULL
)
```

**Arguments**

db_con	a database connection
dir	a directory where we will write the compressed text files output
streamable_table	interface for serializing/deserializing in chunks
lines	the number of lines to use in each single chunk
compress	file compression algorithm. Should be one of "bzip2" (default), "gzip" (faster write times, a bit less compression), "xz", or "none", for no compression.
tables	a list of tables from the database that should be archived. By default, will archive all tables. Table list should specify schema if appropriate, see examples.
method	method to use to query the database, see details.
overwrite	should any existing text files of the same name be overwritten? default is "ask", which will ask for confirmation in an interactive session, and overwrite in a non-interactive script. TRUE will always overwrite, FALSE will always skip such tables.

filter_statement	Typically an SQL "WHERE" clause, specific to your dataset. (e.g., WHERE year = 2013)
filenames	An optional vector of names that will be used to name the files instead of using the tablename from the tables parameter.
callback	An optional function that acts on the data.frame before it is written to disk by streamable_table. It is recommended to use this on a single table at a time. Callback functions must return a data.frame.

## Details

ark will archive tables from a database as (compressed) tsv files. Or other formats that have a streamable\_table method, like parquet. ark does this by reading only chunks at a time into memory, allowing it to process tables that would be too large to read into memory all at once (which is probably why you are using a database in the first place!) Compressed text files will likely take up much less space, making them easier to store and transfer over networks. Compressed plain-text files are also more archival friendly, as they rely on widely available and long-established open source compression algorithms and plain text, making them less vulnerable to loss by changes in database technology and formats.

In almost all cases, the default method should be the best choice. If the `DBI::dbSendQuery()` implementation for your database platform returns the full results to the client immediately rather than supporting chunking with n parameter, you may want to use "window" method, which is the most generic. The "sql-window" method provides a faster alternative for databases like PostgreSQL that support windowing natively (i.e. BETWEEN queries). Note that "window-parallel" only works with streamable\_parquet.

## Value

the path to dir where output files are created (invisibly), for piping.

## Examples

```
# setup
library(dplyr)
dir <- tempdir()
db <- dbplyr::nycflights13_sqlite(tempdir())

## And here we go:
ark(db, dir)

## Not run:

## For a Postgres DB with schema, we can append schema names first
## to each of the table names, like so:
schema_tables <- dbGetQuery(db, sqlInterpolate(db,
  "SELECT table_name FROM information_schema.tables
  WHERE table_schema = ?schema",
  schema = "schema_name"
))

ark(db, dir, tables = paste0("schema_name", ".", schema_tables$table_name))
```

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

---

arkdb_delete_db	<i>delete the local arkdb database</i>
-----------------	--

---

### Description

delete the local arkdb database

### Usage

```
arkdb_delete_db(db_dir = arkdb_dir(), ask = interactive())
```

### Arguments

db_dir	neon database location
ask	Ask for confirmation first?

### Details

Just a helper function that deletes the database files. Usually unnecessary but can be helpful in resetting a corrupt database.

### Examples

```
# Create a db
dir <- tempfile()
db <- local_db(dir)

# Delete it
arkdb_delete_db(dir, ask = FALSE)
```

---

local_db	<i>Connect to a local stand-alone database</i>
----------	--

---

### Description

This function will provide a connection to the best available database. This function is a drop-in replacement for `[DBI::dbConnect]` with behaviour that makes it more subtle for R packages that need a database backend with minimal complexity, as described in details.

**Usage**

```
local_db(
  dbdir = arkdb_dir(),
  driver = Sys.getenv("ARKDB_DRIVER", "duckdb"),
  readonly = FALSE,
  cache_connection = TRUE,
  memory_limit = getOption("duckdb_memory_limit", NA),
  ...
)
```

**Arguments**

<code>dbdir</code>	Path to the database.
<code>driver</code>	Default driver, one of "duckdb", "MonetDBLite", "RSQLite". It will select the first one of those it finds available if a driver is not set. This fallback can be overwritten either by explicit argument or by setting the environmental variable <code>ARKDB_DRIVER</code> .
<code>readonly</code>	Should the database be opened read-only? (duckdb only). This allows multiple concurrent connections (e.g. from different R sessions)
<code>cache_connection</code>	should we preserve a cache of the connection? allows faster load times and prevents connection from being garbage-collected. However, keeping open a read-write connection to duckdb or MonetDBLite will block access of other R sessions to the database.
<code>memory_limit</code>	Set a memory limit for duckdb, in GB. This can also be set for the session by using options, e.g. <code>options(duckdb_memory_limit=10)</code> for a limit of 10GB. On most systems duckdb will automatically set a limit to 80% of machine capacity if not set explicitly.
<code>...</code>	additional arguments (not used at this time)

**Details**

This function provides several abstractions to `[DBI::dbConnect]` to provide a seamless backend for use inside other R packages.

First, this provides a generic method that allows the use of a `[RSQLite::SQLite]` connection if nothing else is available. An argument or environmental variable can be used to override this to manually set a database endpoint for testing purposes.`

Second, this function will cache the database connection in an R environment and load that cache. That means you can call `local_db()` as fast/frequently as you like without causing errors that would occur by rapid calls to `[DBI::dbConnect]`

Third, this function defaults to persistent storage location set by `[tools::R_user_dir]` and configurable by setting the environmental variable `ARKDB_HOME`. This allows a package to provide persistent storage out-of-the-box, and easily switch that storage to a temporary directory (e.g. for testing purposes, or custom user configuration) without having to edit database calls directly.

**Value**

Returns a [DBIconnection] connection to the default database

**Examples**

```
## OPTIONAL: you can first set an alternative home location,  
## such as a temporary directory:  
Sys.setenv(ARKDB_HOME = tempdir())  
  
## Connect to the database:  
db <- local_db()
```

---

local\_db\_disconnect     *Disconnect from the arkdb database.*

---

**Description**

Disconnect from the arkdb database.

**Usage**

```
local_db_disconnect(db = local_db(), env = arkdb_cache)
```

**Arguments**

db                    a DBI connection. By default, will call [local\\_db](#) for the default connection.  
env                    The environment where the function looks for a connection.

**Details**

This function manually closes a connection to the arkdb database.

**Examples**

```
## Disconnect from the database:  
local_db_disconnect()
```

---

process_chunks	<i>process a table in chunks</i>
----------------	----------------------------------

---

**Description**

process a table in chunks

**Usage**

```
process_chunks(  
  file,  
  process_fn,  
  streamable_table = NULL,  
  lines = 50000L,  
  encoding = Sys.getenv("encoding", "UTF-8"),  
  ...  
)
```

**Arguments**

file	path to a file
process_fn	a function of a chunk
streamable_table	interface for serializing/deserializing in chunks
lines	number of lines to read in a chunk.
encoding	encoding to be assumed for input files.
...	additional arguments to streamable_table\$read method.

**Examples**

```
con <- system.file("extdata/mtcars.tsv.gz", package = "arkdb")  
dummy <- function(x) message(paste(dim(x), collapse = " x "))  
process_chunks(con, dummy, lines = 8)
```

---

streamable_base_csv	<i>streamable csv using base R functions</i>
---------------------	--

---

**Description**

streamable csv using base R functions

**Usage**

```
streamable_base_csv()
```



### Details

Follows the comma-separate-values standard using `utils::read.table()`

### Value

a `streamable_table` object (S3)

### See Also

`utils::read.table()`, `utils::write.table()`

---

`streamable_base_tsv`    *streamable tsv using base R functions*

---

### Description

streamable tsv using base R functions

### Usage

```
streamable_base_tsv()
```

### Details

Follows the tab-separate-values standard using `utils::read.table()`, see IANA specification at: <https://www.iana.org/assignments/media-types/text/tab-separated-values>

### Value

a `streamable_table` object (S3)

### See Also

`utils::read.table()`, `utils::write.table()`

streamable\_parquet     *streamable chunked parquet using arrow*

---

**Description**

streamable chunked parquet using arrow

**Usage**

```
streamable_parquet()
```

**Details**

Parquet files are streamed to disk by breaking them into chunks that are equal to the `nlines` parameter in the initial call to `ark`. For each `tablename`, a folder is created and the chunks are placed in the folder in the form `part-000000.parquet`. The software looks at the folder, and increments the name appropriately for the next chunk. This is done intentionally so that users can take advantage of `arrow::open_dataset` in the future, when coming back to review or perform analysis of these data.

**Value**

a `streamable_table` object (S3)

**See Also**

[arrow::read\\_parquet\(\)](#), [arrow::write\\_parquet\(\)](#)

---

streamable\_readr\_csv     *streamable csv using readr*

---

**Description**

streamable csv using readr

**Usage**

```
streamable_readr_csv()
```

**Value**

a `streamable_table` object (S3)

**See Also**

[readr::read\\_csv\(\)](#), [readr::write\\_csv\(\)](#)

---

streamable\_readr\_tsv *streamable tsv using readr*

---

**Description**

streamable tsv using readr

**Usage**

```
streamable_readr_tsv()
```

**Value**

a streamable\_table object (S3)

**See Also**

[readr::read\\_tsv\(\)](#), [readr::write\\_tsv\(\)](#)

---

streamable\_table *streamable table*

---

**Description**

streamable table

**Usage**

```
streamable_table(read, write, extension)
```

**Arguments**

read	read function. Arguments should be "file" (must be able to take a <a href="#">connection()</a> object) and "... " (for) additional arguments.
write	write function. Arguments should be "data" (a data.frame), file (must be able to take a <a href="#">connection()</a> object), and "omit_header" logical, include header (initial write) or not (for appending subsequent chunks)
extension	file extension to use (e.g. "tsv", "csv")

**Details**

Note several constraints on this design. The write method must be able to take a generic R connection object (which will allow it to handle the compression methods used, if any), and the read method must be able to take a `textConnection` object. `readr` functions handle these cases out of the box, so the above method is easy to write. Also note that the write method must be able to `omit_header`. See the built-in methods for more examples.

**Value**

a `streamable_table` object (S3)

**Examples**

```
streamable_readr_tsv <- function() {
  streamable_table(
    function(file, ...) readr::read_tsv(file, ...),
    function(x, path, omit_header) {
      readr::write_tsv(x = x, path = path, omit_header = omit_header)
    },
    "tsv"
  )
}
```

---

<code>streamable_vroom</code>	<i>streamable tables using vroom</i>
-------------------------------	--------------------------------------

---

**Description**

streamable tables using vroom

**Usage**

```
streamable_vroom()
```

**Value**

a `streamable_table` object (S3)

**See Also**

[readr::read\\_tsv\(\)](#), [readr::write\\_tsv\(\)](#)

---

<code>unark</code>	<i>Unarchive a list of compressed tsv files into a database</i>
--------------------	---

---

**Description**

Unarchive a list of compressed tsv files into a database

**Usage**

```

unark(
  files,
  db_con,
  streamable_table = NULL,
  lines = 50000L,
  overwrite = "ask",
  encoding = Sys.getenv("encoding", "UTF-8"),
  tablenames = NULL,
  try_native = TRUE,
  ...
)

```

**Arguments**

<code>files</code>	vector of filenames to be read in. Must be tsv format, optionally compressed using bzip2, gzip, zip, or xz format at present.
<code>db_con</code>	a database src ( <code>src_dbi</code> object from <code>dplyr</code> )
<code>streamable_table</code>	interface for serializing/deserializing in chunks
<code>lines</code>	number of lines to read in a chunk.
<code>overwrite</code>	should any existing text files of the same name be overwritten? default is "ask", which will ask for confirmation in an interactive session, and overwrite in a non-interactive script. TRUE will always overwrite, FALSE will always skip such tables.
<code>encoding</code>	encoding to be assumed for input files.
<code>tablenames</code>	vector of tablenames to be used for corresponding files. By default, tables will be named using lowercase names from file basename with special characters replaced with underscores (for SQL compatibility).
<code>try_native</code>	logical, default TRUE. Should we try to use a native bulk import method for the database connection? This can substantially speed up read times and will fall back on the DBI method for any table that fails to import. Currently only MonetDBLite connections support this.
<code>...</code>	additional arguments to <code>streamable_table\$read</code> method.

**Details**

`unark` will read in a files in chunks and write them into a database. This is essential for processing large compressed tables which may be too large to read into memory before writing into a database. In general, increasing the `lines` parameter will result in a faster total transfer but require more free memory for working with these larger chunks.

If using `readr`-based `streamable-table`, you can suppress the progress bar by using `options(readr.show_progress = FALSE)` when reading in large files.

**Value**

the database connection (invisibly)

**Examples**

```
## Setup: create an archive.
library(dplyr)
dir <- tempdir()
db <- dbplyr::nycflights13_sqlite(tempdir())

## database -> .tsv.bz2
ark(db, dir)

## list all files in archive (full paths)
files <- list.files(dir, "bz2$", full.names = TRUE)

## Read archived files into a new database (another sqlite in this case)
new_db <- DBI::dbConnect(RSQLite::SQLite())
unark(files, new_db)

## Prove table is returned successfully.
tbl(new_db, "flights")
```

# Index

ark, [3](#)  
ark(), [2](#)  
arkdb (arkdb-package), [2](#)  
arkdb-package, [2](#)  
arkdb\_delete\_db, [5](#)  
arrow::read\_parquet(), [10](#)  
arrow::write\_parquet(), [10](#)  
  
connection(), [11](#)  
  
DBI::dbSendQuery(), [4](#)  
duckdb::duckdb, [6](#)  
  
local\_db, [5, 7](#)  
local\_db\_disconnect, [7](#)  
  
process\_chunks, [8](#)  
  
readr::read\_csv(), [10](#)  
readr::read\_tsv(), [11, 12](#)  
readr::write\_csv(), [10](#)  
readr::write\_tsv(), [11, 12](#)  
  
streamable\_base\_csv, [8](#)  
streamable\_base\_tsv, [9](#)  
streamable\_parquet, [10](#)  
streamable\_readr\_csv, [10](#)  
streamable\_readr\_tsv, [11](#)  
streamable\_table, [11](#)  
streamable\_vroom, [12](#)  
  
unark, [12](#)  
unark(), [2](#)  
utils::read.table(), [9](#)  
utils::write.table(), [9](#)