

# Package ‘rbedrock’

March 25, 2025

**Title** Analysis and Manipulation of Data from Minecraft Bedrock Edition

**Version** 0.3.3

**Description** Implements an interface to Minecraft (Bedrock Edition) worlds. Supports the analysis and management of these worlds and game saves.

**License** MIT + file LICENSE

**Encoding** UTF-8

**Language** en-US

**RoxygenNote** 7.3.2

**SystemRequirements** cmake, zlib, GNU make, Solaris: g++ is a requirement

**NeedsCompilation** yes

**Depends** R (>= 3.6.0)

**Imports** tibble, R6, stringr, bit64, rappdirs, rlang, dplyr, purrr, magrittr, readr, utils, digest, vctrs, tidyr, fs

**Suggests** zip, testthat, jsonlite, covr

**URL** <https://github.com/reedacartwright/rbedrock>

**BugReports** <https://github.com/reedacartwright/rbedrock/issues>

**Author** Reed Cartwright [aut, cre] (<<https://orcid.org/0000-0002-0837-9380>>),  
Rich FitzJohn [ctb],  
Christian Stigen Larsen [ctb],  
The LevelDB Authors [cph]

**Maintainer** Reed Cartwright <[racartwright@gmail.com](mailto:racartwright@gmail.com)>

**Repository** CRAN

**Date/Publication** 2025-03-25 22:30:02 UTC

## Contents

ActorDigest . . . . .	2
Actors . . . . .	3

bedrockdb . . . . .	4
bedrock_random . . . . .	6
bedrock_random_create_seed . . . . .	7
Biomes . . . . .	8
BlockEntity . . . . .	9
Checksums . . . . .	10
ChunkVersion . . . . .	11
chunk_blocks . . . . .	12
chunk_keys . . . . .	13
chunk_origin . . . . .	14
Data2D . . . . .	14
Data3D . . . . .	16
delete_values . . . . .	17
Entity . . . . .	18
FinalizedState . . . . .	19
get_chunk_blocks_data . . . . .	20
get_keys . . . . .	22
get_nbt_data . . . . .	22
get_values . . . . .	24
HSA . . . . .	24
list_biomes . . . . .	26
locate_blocks . . . . .	26
minecraft_worlds . . . . .	27
nbt_byte . . . . .	28
PendingTicks . . . . .	29
put_values . . . . .	30
RandomTicks . . . . .	31
rbedrock_example . . . . .	32
read_leveldat . . . . .	32
simulation_area . . . . .	33
spawning_area . . . . .	33
SubchunkBlocks . . . . .	34
<b>Index</b>	<b>37</b>

---

 ActorDigest

*Read and write Actor Digest Data*


---

## Description

Actor digests store a list of all entities in a chunk; however they are not chunk data and use their own prefix. The key format for actor digest data is `acdig:x:z:dimension`.

`get_acdig_data()` and `get_acdig_value()` load ActorDigest data from db. `get_acdig_value()` supports loading only a single value.

`put_acdig_data()` and `put_acdig_value()` store ActorDigest data into db.

`read_acdig_value()` and `write_acdig_value()` decode and encode ActorDigest data respectively.

`create_acdig_keys()` creates keys for ActorDigest data.

### Usage

```
get_acdig_data(x, z, dimension, db)
get_acdig_value(x, z, dimension, db)
put_acdig_data(values, x, z, dimension, db)
put_acdig_value(value, x, z, dimension, db)
read_acdig_value(rawdata)
write_acdig_value(value)
create_acdig_keys(x, z, dimension)
```

### Arguments

<code>x, z, dimension</code>	Chunk coordinates to extract data from. <code>x</code> can also be a character vector of db keys.
<code>db</code>	A bedrockdb object.
<code>values</code>	A list of character vectors. If <code>x</code> is missing, the names of values will be taken as the keys.
<code>value</code>	A character vector.
<code>rawdata</code>	A raw vector.

### Value

`get_acdig_values()` returns a vector of actor keys. `get_acdig_data()` returns a named list of the of the values returned by `get_acdig_value()`.

### See Also

[Actors, Entity](#)

### Description

After 1.18.30, the nbt data of each actor is saved independently in the database, using a key with a prefix and a 16-character storage key: 'actor:0123456789abcdef'. The keys of all actors in a chunk are saved in an [ActorDigest](#) record, with format `acdig:x:z:dimension`.

## Usage

```
get_actors_data(x, z, dimension, db)
get_actors_value(x, z, dimension, db)
put_actors_data(values, x, z, dimension, db)
put_actors_value(value, x, z, dimension, db)
```

## Arguments

x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
db	A bedrockdb object.
values	A list of character vectors. If x is missing, the names of values will be taken as the keys.
value	A list of nbt actors data

## Details

`get_actors_value()` loads Actors data for a single chunk in db. `get_actors_data()` loads Actors data from multiple chunks in db.

`put_actors_value()` and `put_actors_data()` store one/multiple chunks Actors data into db and update the chunks' ActorDigests. When storing Actors data, an actor's storage key will be recalculated from the actor's UniqueID. The actor's position and dimension are not verified to be in the chunk it is assigned to.

## See Also

[ActorDigest](#), [Entity](#)

---

bedrockdb

*Open a Bedrock Edition world for reading and writing.*

---

## Description

bedrockdb opens a handle to a leveldb database that contains save-game data for a Bedrock Edition world. On success, it returns an R6 class of type 'bedrockdb' that can be used directly for low-level reading and writing access to the db or can be passed to higher-level functions. The handle to the database can be closed by passing it to `close`.

**Usage**

```
bedrockdb(
  path,
  create_if_missing = FALSE,
  error_if_exists = NULL,
  paranoid_checks = NULL,
  write_buffer_size = 4194304L,
  max_open_files = NULL,
  block_size = 163840L,
  cache_capacity = 83886080L,
  bloom_filter_bits_per_key = 10L,
  compression_level = -1L
)
```

```
## S3 method for class 'bedrockdb'
close(con, compact = FALSE, ...)
```

```
is_bedrockdb(x)
```

**Arguments**

path	The path to a world folder. If the path does not exist, it is assumed to be the base name of a world folder in the local minecraftWorlds directory.
create_if_missing	Create world database if it doesn't exist.
error_if_exists	Raise an error if the world database already exists.
paranoid_checks	Internal leveldb option
write_buffer_size	Internal leveldb option
max_open_files	Internal leveldb option
block_size	Internal leveldb option
cache_capacity	Internal leveldb option
bloom_filter_bits_per_key	Internal leveldb option
compression_level	Internal leveldb option
con	An database object created by bedrockdb.
compact	Compact database before closing.
...	arguments passed to or from other methods.
x	An object.

**Value**

On success, bedrockdb returns an R6 class of type 'bedrockdb'.

**Examples**

```
# open an example works and get all keys
dbpath <- rbedrock_example_world("example1.mcworld")
db <- bedrockdb(dbpath)
keys <- get_keys(db)
close(db)

## Not run:

# open a world in the minecraftWorlds folder using a world id.
db <- bedrockdb("lrkkYFpUABA=")
# do something with db ...
close(db)

# open a world using absolute path
db <- bedrockdb("C:\\\\minecraftWorlds\\\\my_world")
# do something with db ...
close(db)

## End(Not run)
```

---

bedrock\_random

*Random Number Generation for Minecraft*


---

**Description**

Bedrock Edition's central random number algorithm is MT19937. However, R's MT19937 code is not compatible with Bedrock's. These routines provide an API that is compatible with Bedrock's.

bedrock\_random\_seed() seeds the random number generator.

bedrock\_random\_state() returns the current state of the random number generator as a raw vector.

bedrock\_random\_get\_uint() returns a 32-bit random integer. Default range is  $[0, 2^{32}-1]$ .

bedrock\_random\_get\_int() returns a 31-bit random integer. Default range is  $[0, 2^{31}-1]$ .

bedrock\_random\_get\_float() returns a random real number. Default range is  $[0.0, 1.0)$ .

bedrock\_random\_get\_double() returns a random real number. Default range is  $[0.0, 1.0)$ .

**Usage**

```
bedrock_random_seed(value)
```

```
bedrock_random_state(new_state = NULL)
```

```
bedrock_random_get_uint(n, max)
```

```
bedrock_random_get_int(n, min, max)
```

```
bedrock_random_get_float(n, min, max)
```

```
bedrock_random_get_double(n)
```

**Arguments**

value	a scalar integer
new_state	a raw vector
n	number of observations.
min, max	lower and upper limits of the distribution. Must be finite. If only one is specified, it is taken as max. If neither is specified, the default range is used.

**Examples**

```
# seed the global random number generator
bedrock_random_seed(5490L)

# save and restore rng state
saved_state <- bedrock_random_state()
bedrock_random_get_uint(10)
bedrock_random_state(saved_state)
bedrock_random_get_uint(10)
```

---

bedrock\_random\_create\_seed

*Random Number Seeds for Minecraft*

---

**Description**

bedrock\_random\_create\_seed() constructs a seed using the formulas type 1:  $x*a + z*b + salt$ , type 2:  $x*a + z*b + salt$ , and type 3:  $x*a + z*b + salt$ .

**Usage**

```
bedrock_random_create_seed(x, z, a, b, salt, type)
```

**Arguments**

x, z	chunk coordinates
a, b	seed parameters
salt	seed parameter
type	which seed type to use

**Details**

Minecraft uses several different kind of seeds during world generation and gameplay.

**Examples**

```
# identify slime chunks
g <- tidyr::expand_grid(x=1:10, z=1:10)
is_slime_chunk <- purrr::pmap_lgl(g, function(x,z) {
  seed <- bedrock_random_create_seed(x,z,0x1f1f1f1f,1,0,type=1)
  bedrock_random_seed(seed)
  bedrock_random_get_uint(1,10) == 0
})
```

---

Biomes

*Read and write biomes data.*


---

**Description**

Biomes data is stored as the second map in Data3D data (tag 43). Legacy Biomes data is stored as the second map in the Data2D data (tag 45).

`get_biomes_data()` and `get_biomes_value()` load Biomes data from db. `get_biomes_data()` will silently drop keys not representing Data3D data. `get_biomes_value()` supports loading only a single value. `get_biomes_values()` is a synonym for `get_biomes_data()`.

`put_biomes_data()` `put_biomes_values()`, and `put_biomes_value()` update the biome information of chunks. They preserve any existing height data.

`get_legacy_biomes_*`() and `put_legacy_biomes_*`() behave similar to the equivalent non-legacy functions. They get or put 2d biome data.

**Usage**

```
get_biomes_data(db, x, z, dimension, return_names = TRUE)
```

```
get_biomes_values(db, x, z, dimension, return_names = TRUE)
```

```
get_biomes_value(db, x, z, dimension, return_names = TRUE)
```

```
put_biomes_data(db, data, missing_height = -64L)
```

```
put_biomes_values(db, x, z, dimension, values, missing_height = -64L)
```

```
put_biomes_value(db, x, z, dimension, value, missing_height = -64L)
```

```
get_legacy_biomes_data(db, x, z, dimension, return_names = TRUE)
```

```
get_legacy_biomes_values(db, x, z, dimension, return_names = TRUE)
```

```
get_legacy_biomes_value(db, x, z, dimension, return_names = TRUE)
```

```
put_legacy_biomes_data(db, data, missing_height = 0L)
```

```
put_legacy_biomes_values(db, x, z, dimension, values, missing_height = 0L)
```



```
put_legacy_biomes_value(db, x, z, dimension, value, missing_height = 0L)
```

### Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
return_names	return biome names instead of biome ids.
data	A list of character or integer vectors.
missing_height	if there is no existing height data, use this value for the chunk.
values	a list of arrays containing biome names or ids.
value	an array containing biome names or ids.

### Value

get\_biomes\_value() returns an array with 3 dimensions. get\_biomes\_data() returns a list of the of the values returned by get\_biomes\_value().

---

BlockEntity

*Load and store BlockEntity NBT data*

---

### Description

BlockEntity data (tag 49) holds a list of NBT values for entity data associated with specific blocks.

get\_block\_entity\_data() and get\_block\_entity\_value() load BlockEntity data from db. get\_block\_entity\_data() will silently drop and keys not representing BlockEntity data. get\_block\_entity\_value() supports loading only a single value. get\_block\_entity\_values() is a synonym for get\_block\_entity\_data().

put\_block\_entity\_values(), put\_block\_entity\_value(), and put\_block\_entity\_data() store BlockEntity data into db.

### Usage

```
get_block_entity_data(db, x, z, dimension)
```

```
get_block_entity_values(db, x, z, dimension)
```

```
get_block_entity_value(db, x, z, dimension)
```

```
put_block_entity_values(db, x, z, dimension, values)
```

```
put_block_entity_value(db, x, z, dimension, value)
```

```
put_block_entity_data(db, data)
```

**Arguments**

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
values	A list of nbt objects
value	An nbt object.
data	A named-list specifying key-value pairs.

**Value**

`get_block_entity_data()` returns a named-list of nbt data. `get_block_entity_values()` returns a single nbt value.

---

Checksums	<i>Load and store Checksums data</i>
-----------	--------------------------------------

---

**Description**

Checksums data (tag 59) holds checksums for several chunk records. These records are 2DMaps (tag 45), SubchunkBlocks (tag 47), BlockEntities (tag 49), and Entities (tag 50).

`get_checksums_data()` loads Checksums data from a bedrockdb. It will silently drop and keys not representing Checksums data. `get_checksums_values()` is a synonym for `get_checksums_data()`.

`get_checksums_value()` loads Checksums data from a bedrockdb. It only supports loading a single value.

`update_checksums_data()` recalculates Checksums data. It calculates checksums for the specified chunks' SubchunkBlocks, 2DMaps, BlockEntities, and Entities records in db and updates the Checksums record to match.

`read_checksums_value()` parses a binary Checksums record into a list of checksums.

`write_checksums_value()` converts Checksums from a named list into binary format.

**Usage**

```
get_checksums_data(db, x, z, dimension)
```

```
get_checksums_values(db, x, z, dimension)
```

```
get_checksums_value(db, x, z, dimension)
```

```
update_checksums_data(db, x, z, dimension)
```

```
read_checksums_value(rawdata)
```

```
write_checksums_value(object)
```

**Arguments**

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
rawdata	a raw vector holding binary Checksums data
object	a named character vector in the same format as returned by read_checksums_value().

**Value**

get\_checksums\_data() returns a named-list of the values returned by get\_checksums\_value().  
 get\_checksums\_value() and read\_checksums\_value() return a character vector. The names of the character vector indicate which chunk record (tag and subtag) the checksum is for.  
 write\_checksums\_value() returns a raw vector.

---

 ChunkVersion

*Read and write chunk version data*


---

**Description**

Version data (tag 44) and LegacyVersion data (tag 118) store the version number of a chunk. In Minecraft version 1.16.100, chunk version data was moved from tag 118 to tag 44.

get\_chunk\_version\_data() and get\_chunk\_version\_value() load Version data from db. get\_chunk\_version\_data() will silently drop and keys not representing Version data. get\_chunk\_version\_value() supports loading only a single value. get\_chunk\_version\_values() is a synonym for get\_chunk\_version\_data().

put\_chunk\_version\_data(), put\_chunk\_version\_values(), and put\_chunk\_version\_value() store Version data into a bedrockdb.

read\_chunk\_version\_value() decodes Version data.

write\_chunk\_version\_value() encodes Version data.

**Usage**

```
get_chunk_version_data(db, x, z, dimension)
```

```
get_chunk_version_values(db, x, z, dimension)
```

```
get_chunk_version_value(db, x, z, dimension)
```

```
put_chunk_version_data(db, data)
```

```
put_chunk_version_values(db, x, z, dimension, values)
```

```
put_chunk_version_value(db, x, z, dimension, value)
```

```
read_chunk_version_value(rawdata)
```

```
write_chunk_version_value(num)
```

**Arguments**

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract version data from. x can also be a character vector of db keys.
data	A named-vector of key-value pairs for Version data.
values	An integer vector
value	A scalar integer vector
rawdata	A scalar raw.
num	A scalar integer.

---

chunk_blocks	<i>Extract or replace chunk blocks from an array</i>
--------------	--

---

**Description**

Convenience wrappers around `[]` to extract or replace blocks from an array based on block coordinates.

**Usage**

```
chunk_blocks(x, ..., drop = TRUE, origin = chunk_origin(x))
```

```
chunk_blocks(x, ..., origin = chunk_origin(x)) <- value
```

**Arguments**

x	Object from which to extract element(s) or in which to replace element(s).
...	block coordinates specifying elements to extract or replace. Can be numeric, logical, or missing. If numeric, the coordinates will be mapped to indices unless there is a single, non-matrix argument.
drop	if TRUE the result is coerced to the lowest possible dimension.
origin	the origin of the chunk array, used for mapping coordinates to indices
value	An array-like R object of similar class as x

**Description**

Chunk keys are keys to chunk data. A chunk key has a format which indicates the chunk it holds data for and the type of data it holds. This format is either `chunk:x:z:d:t` or `chunk:x:z:d:t:s`, where `x` and `z` indicates the coordinates of the chunk in chunk space, `d` indicates the dimension of the chunk, and `t` and `s` indicate the tag and subtag of the chunk.

`parse_chunk_keys()` splits chunk keys into their individual elements and returns a table with the results. Keys that do not contain chunk data are silently dropped.

`create_chunk_keys()` returns a vector of chunk keys formed from its arguments.

`chunk_positions()` returns a matrix containing the chunk coordinates of keys.

`chunk_origins()` returns a matrix containing the block coordinate of the NW corner of keys.

`chunk_tag_str()` and `chunk_tag_int()` convert between integer and character representations of chunk tags.

**Usage**

```
parse_chunk_keys(keys)
```

```
create_chunk_keys(x, z, dimension, tag, subtag)
```

```
chunk_positions(keys)
```

```
chunk_origins(keys)
```

```
chunk_tag_str(tags)
```

```
chunk_tag_int(tags)
```

**Arguments**

<code>keys</code>	A character vector of database keys.
<code>x</code>	Chunk x-coordinate.
<code>z</code>	Chunk z-coordinate.
<code>dimension</code>	Dimension.
<code>tag</code>	The type of chunk data.
<code>subtag</code>	The subchunk the key refers to (Only used for tag 47).
<code>tags</code>	a vector

**Examples**

```

parse_chunk_keys("chunk:0:0:0:44")
parse_chunk_keys("chunk:0:0:0:47:1")
create_chunk_keys(0, 0, 0, 47, 1)

```

---

chunk_origin	<i>Get or set the coordinates of the origin of a chunk</i>
--------------	--

---

**Description**

Get or set the coordinates of the origin of a chunk

**Usage**

```

chunk_origin(x)

chunk_origin(x) <- value

```

**Arguments**

x	an array of block data
value	an integer vector

---

Data2D	<i>Read and write Data2D data</i>
--------	-----------------------------------

---

**Description**

Data2D data (tag 45) stores information about surface heights and biomes in a chunk. Data2D data is 768 bytes long and consists of a 256 int16s (heights) followed by 256 uint8s (biomes).

get\_data2d\_data() loads Data2D data from a bedrockdb. It will silently drop and keys not representing Data2D data.

get\_data2d\_value() loads Data2D data from a bedrockdb. It only supports loading a single value.

read\_data2d\_value decodes binary Data2D data.

put\_data2d\_data(), put\_data2d\_values(), and put\_data2d\_value() store Data2D data into a bedrockdb.

write\_data2d\_value encodes Data2D data into a raw vector.

**Usage**

```

get_data2d_data(db, x, z, dimension)

get_data2d_values(db, x, z, dimension)

get_data2d_value(db, x, z, dimension)

read_data2d_value(rawdata)

put_data2d_data(db, data)

put_data2d_values(db, x, z, dimension, height_maps, biome_maps)

put_data2d_value(db, x, z, dimension, height_map, biome_map)

write_data2d_value(height_map, biome_map)

```

**Arguments**

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
rawdata	A raw vector.
data	A named-vector of key-value pairs for Data2D data.
height_maps, biome_maps	Lists of height and biome data. Values will be recycled if necessary to match the number of keys to be written to. If biome_maps is missing, height_maps should be in the same format as returned by get_data2d_data().
height_map, biome_map	16x16 arrays containing height and biome data. Values will be recycled if necessary. If biome_map is missing, height-map should be a list a list() with both "height_map" and "biome_map" elements.

**Value**

get\_data2d\_data() returns a list of the of the values returned by get\_data2d\_value().  
get\_data2d\_value() returns a list with components "height\_map" and "biome\_map".

**Examples**

```

heights <- matrix(63,16,16)
biomes <- matrix(1,16,16)
# Pass heights and biomes as separate parameters
dat <- write_data2d_value(heights, biomes)
# Pass them as a list.
obj <- list(height_map = heights, biome_map = biomes)
dat <- write_data2d_value(obj)
# Pass them as scalars

```

```
dat <- write_data2d_value(63, 1)
```

---

 Data3D

*Read and write Data3D data*


---

## Description

Data3D data (tag 43) stores information about surface heights and biomes in a chunk.

`get_data3d_data()` loads Data3D data from db. It will silently drop keys not representing Data3D data.

`get_data3d_value()` loads Data3D data from db. It only supports loading a single value.

`put_data3d_data()`, `put_data3d_values()`, and `put_data3d_value()` store Data3D data into db.

`read_data3d_value()` decodes binary Data3D data.

`write_data3d_value` encodes Data3D data into a raw vector.

## Usage

```
get_data3d_data(db, x, z, dimension)
```

```
get_data3d_values(db, x, z, dimension)
```

```
get_data3d_value(db, x, z, dimension)
```

```
put_data3d_data(db, data)
```

```
put_data3d_values(db, x, z, dimension, height_maps, biome_maps)
```

```
put_data3d_value(db, x, z, dimension, height_map, biome_map)
```

```
read_data3d_value(rawdata)
```

```
write_data3d_value(height_map, biome_map)
```

## Arguments

`db` A bedrockdb object.

`x, z, dimension` Chunk coordinates to extract data from. `x` can also be a character vector of db keys.

`data` A named-vector of key-value pairs for Data3D data.

`height_maps, biome_maps`

Lists of height and biome data. Values will be recycled if necessary to match the number of keys to be written to. If `biome_maps` is missing, `height_maps` should be in the same format as returned by `get_data3d_data()`.



height_map	16x16 array containing height data. Values will be recycled if necessary. If biome_map is missing, height-map should be a list a list() with both "height_map" and "biome_map" elements.
biome_map	16xNx16 array containing biome data.
rawdata	A raw vector.

**Value**

get\_data3d\_data() returns a list of the of the values returned by get\_data3d\_value().

get\_data3d\_value() returns a list with components "height\_map" and "biome\_map".

---

delete_values	<i>Remove values from a bedrockdb.</i>
---------------	--

---

**Description**

Remove values from a bedrockdb.

**Usage**

```
delete_values(
  db,
  keys,
  report = FALSE,
  readoptions = NULL,
  writeoptions = NULL
)
```

**Arguments**

db	A bedrockdb object
keys	A character vector of keys.
report	A logical indicating whether to generate a report on deleted keys
readoptions	A bedrock_leveldb_readoptions object
writeoptions	A bedrock_leveldb_writeoptions object

**Value**

If report == TRUE, a logical vector indicating which keys were deleted.

---

 Entity

*Load and store Entity NBT data*


---

### Description

Entity data (tag 50) holds a list of NBT values for mobs and other entities in the game. After 1.18.30, entity data was migrated to a new actor digest format and no longer saved with chunk data.

`get_entity_data()` and `get_entity_value()` load Entity data from db. `get_entity_data()` will silently drop and keys not representing Entity data. `get_entity_value()` supports loading only a single value. `get_entity_values()` is a synonym for `get_entity_data()`.

`put_entity_values()`, `put_entity_value()`, and `put_entity_data()` store BlockEntity data into db.

### Usage

```
get_entity_data(db, x, z, dimension)
```

```
get_entity_values(db, x, z, dimension)
```

```
get_entity_value(db, x, z, dimension)
```

```
put_entity_values(db, x, z, dimension, values)
```

```
put_entity_value(db, x, z, dimension, value)
```

```
put_entity_data(db, data)
```

### Arguments

<code>db</code>	A bedrockdb object.
<code>x, z, dimension</code>	Chunk coordinates to extract data from. <code>x</code> can also be a character vector of db keys.
<code>values</code>	A list of nbt objects
<code>value</code>	An nbt object.
<code>data</code>	A named-list specifying key-value pairs.

### Value

`get_entity_data()` returns a named-list of nbt data. `get_entity_values()` returns a single nbt value.

---

FinalizedState	<i>Load and store FinalizedState data</i>
----------------	---

---

### Description

FinalizedState data (tag 54) holds a number which indicates a chunk's state of generation.

`get_finalized_state_data()` and `get_finalized_state_value()` load FinalizedState data from db. `get_finalized_state_data()` will silently drop and keys not representing FinalizedState data. `get_finalized_state_value()` supports loading only a single value. `get_finalized_state_values()` is a synonym for `get_finalized_state_data()`.

`put_finalized_state_data()`, `put_finalized_state_values()`, and `put_finalized_state_value()` store FinalizedState data into a bedrockdb.

`read_finalized_state_value()` parses a binary FinalizedState record.

`write_finalized_state_value()` converts a FinalizedState value to a raw vector.

### Usage

```
get_finalized_state_data(db, x, z, dimension)
```

```
get_finalized_state_values(db, x, z, dimension)
```

```
get_finalized_state_value(db, x, z, dimension)
```

```
put_finalized_state_data(db, data)
```

```
put_finalized_state_values(db, x, z, dimension, values)
```

```
put_finalized_state_value(db, x, z, dimension, value)
```

```
read_finalized_state_value(rawdata)
```

```
write_finalized_state_value(value)
```

### Arguments

<code>db</code>	A bedrockdb object.
<code>x, z, dimension</code>	Chunk coordinates to extract data from. <code>x</code> can also be a character vector of db keys.
<code>data</code>	A named-vector of key-value pairs for FinalizedState data.
<code>values</code>	An integer vector
<code>value</code>	a scalar integer
<code>rawdata</code>	a raw vector

**Details**

FinalizedState data contains the following information.

Value	Name	Description
0	NeedsInstaticking	Chunk needs to be ticked
1	NeedsPopulation	Chunk needs to be populated with mobs
2	Done	Chunk generation is fully complete

**Value**

get\_finalized\_state\_data() returns a named integer vector of the values returned by get\_finalized\_state\_value().

---

get\_chunk\_blocks\_data *Load block data from one or more chunks*

---

**Description**

These functions return block data as strings containing the block name and block states. The strings' format is blockname@state1=value1@state2=value2 etc. Blocks may have 0 or more states.

get\_chunk\_blocks\_value() is an alias for get\_chunk\_blocks\_data()

get\_chunk\_blocks\_value() loads block data from a bedrockdb. It only supports loading a single value.

put\_chunk\_blocks\_data(), put\_chunk\_blocks\_values(), and put\_chunk\_blocks\_value() stores block data into a bedrockdb.

**Usage**

```
get_chunk_blocks_data(
  db,
  x,
  z,
  dimension,
  names_only = FALSE,
  extra_block = !names_only,
  min_subchunk = TRUE,
  max_subchunk = !names_only
)
```

```
get_chunk_blocks_values(
  db,
  x,
  z,
  dimension,
  names_only = FALSE,
  extra_block = !names_only,
```

```

    min_subchunk = TRUE,
    max_subchunk = !names_only
)

get_chunk_blocks_value(
  db,
  x,
  z,
  dimension,
  names_only = FALSE,
  extra_block = !names_only,
  min_subchunk = TRUE,
  max_subchunk = !names_only
)

put_chunk_blocks_data(db, data, version = 9L)

put_chunk_blocks_values(db, x, z, dimension, values, version = 9L)

put_chunk_blocks_value(db, x, z, dimension, value, version = 9L)

```

### Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.
names_only	A logical scalar. Return only the names of the blocks, ignoring block states.
extra_block	A logical scalar. Append the extra block layer to the output (separated by ";"). This is mostly useful if you have waterlogged blocks. If the extra block is air, it will not be appended.
min_subchunk, max_subchunk	The minimum and maximum subchunks of the returned array. Set to TRUE to use dimension defaults. Set to FALSE to use the values from chunk data.
data	A named list of 16xNx16 character() arrays
version	Which format of subchunk data to use
values	A list of 16xNx16 character() arrays
value	A 16xNx16 character array

### Value

get\_chunk\_blocks\_data() returns a list of the of the values returned by read\_chunk\_blocks\_value().

get\_chunk\_blocks\_value() return a 16xNx16 character array. The axes represent the x, y, and z dimensions in that order. The size of the y-axis is based on the highest subchunk in the coordinate. Missing subchunks are considered air.

---

get_keys	<i>Get a list of keys stored in a bedrockdb.</i>
----------	--

---

### Description

Get a list of keys stored in a bedrockdb.

### Usage

```
get_keys(db, starts_with = NULL, readoptions = NULL)
```

### Arguments

db	A bedrockdb object
starts_with	A string specifying chunk prefix or string prefix.
readoptions	A bedrock_leveldb_readoptions object

### Value

A vector containing all the keys found in the bedrockdb.

If starts\_with is specified, this vector will be filtered for based on the specified prefix.

---

get_nbt_data	<i>Read and Write NBT Data</i>
--------------	--------------------------------

---

### Description

get\_nbt\_data() and get\_nbt\_value() load nbt-formatted data from db and parses it. get\_nbt\_values() is a synonym for get\_nbt\_data().

put\_nbt\_values(), put\_nbt\_value(), and put\_nbt\_data() store nbt data into db in binary form.

read\_nbt reads NBT data from a raw vector.

read\_nbt\_data calls read\_nbt on each element of a list.

write\_nbt encodes NBT data into a raw vector.

write\_nbt\_data calls write\_nbt on each element of a list.

**Usage**

```
get_nbt_data(db, keys, readoptions = NULL, simplify = TRUE)
get_nbt_value(db, key, readoptions = NULL, simplify = TRUE)
get_nbt_values(db, keys, readoptions = NULL, simplify = TRUE)
put_nbt_values(db, keys, values, writeoptions = NULL)
put_nbt_value(db, key, value, writeoptions = NULL)
put_nbt_data(db, data, writeoptions = NULL)
read_nbt(rawdata, simplify = TRUE)
read_nbt_data(data, simplify = TRUE)
write_nbt(object)
write_nbt_data(data)
```

**Arguments**

<code>db</code>	A bedrockdb object
<code>keys</code>	A character vector of keys.
<code>readoptions</code>	A <code>bedrock_leveldb_readoptions</code> object
<code>simplify</code>	If TRUE, simplifies a list containing a single unnamed <code>nbtnode</code> .
<code>key</code>	A single key.
<code>values</code>	A list of <code>nbt</code> objects
<code>writeoptions</code>	A <code>bedrock_leveldb_writeoptions</code> object
<code>value</code>	An <code>nbt</code> object.
<code>data</code>	A named-list specifying key-value pairs.
<code>rawdata</code>	A raw vector
<code>object</code>	An <code>nbt</code> object or a list of <code>nbt</code> objects

**Details**

The Named Binary Tag (NBT) format is used by Minecraft for various data types.

---

get_values	<i>Read values stored in a bedrockdb.</i>
------------	---

---

**Description**

get\_values() and get\_data() are synonyms.

**Usage**

```
get_values(db, keys, starts_with, readoptions = NULL)
```

```
get_data(db, keys, starts_with, readoptions = NULL)
```

```
get_value(db, key, readoptions = NULL)
```

```
has_values(db, keys, readoptions = NULL)
```

**Arguments**

db	A bedrockdb object
keys	A character vector of keys.
starts_with	A string specifying chunk prefix or string prefix.
readoptions	A bedrock_leveldb_readoptions object
key	A single key.

**Value**

get\_values() returns a named-list of raw vectors.

get\_value() returns a raw vector.

has\_values() returns a logical vector.

---

HSA	<i>Read and write HardcodedSpawnArea (HSA) data</i>
-----	---

---

**Description**

HardcodedSpawnArea (HSA) data (tag 57) stores information about any structure spawning locations in a chunk. An HSA is defined by a bounding box that specifies the location of an HSA in a chunk and a tag that specifies the type: 1 = NetherFortress, 2 = SwampHut, 3 = OceanMonument, and 5 = PillagerOutpost.

get\_hsa\_data() loads HardcodedSpawnArea data from a bedrockdb. It will silently drop and keys not representing HSA data. get\_hsa\_values() is a synonym for get\_hsa\_data().

get\_hsa\_value() loads HSA data from a bedrockdb. It only supports loading a single value.



`read_hsa_value()` decodes HSA data.

`put_hsa_data()` puts HSA data into a bedrockdb. HSA bounding boxes will be split across chunks and

`put_hsa_values()` and `put_hsa_value()` store HSA data into a bedrockdb.

`write_hsa_value()` encodes HSA data.

### Usage

```
get_hsa_data(db, x, z, dimension)
```

```
get_hsa_values(db, x, z, dimension)
```

```
get_hsa_value(db, x, z, dimension)
```

```
read_hsa_value(rawdata)
```

```
put_hsa_data(db, data, merge = TRUE)
```

```
put_hsa_values(db, x, z, dimension, values)
```

```
put_hsa_value(db, x, z, dimension, value)
```

```
write_hsa_value(value)
```

### Arguments

<code>db</code>	A bedrockdb object.
<code>x, z, dimension</code>	Chunk coordinates to extract data from. <code>x</code> can also be a character vector of db keys.
<code>rawdata</code>	A scalar raw.
<code>data</code>	A table containing HSA coordinates.
<code>merge</code>	Merge the new HSAs with existing HSAs.
<code>values</code>	A list of tables containing HSA coordinates and tags.
<code>value</code>	A table containing HSA coordinates

### Value

`get_hsa_data()` returns a table in the same format as `get_hsa_value()`.

`get_hsa_value()` and `read_hsa_value()` return a table with columns indicating the coordinates of the HSA bounding box and the location of the HSS at the center of the bounding box. `get_hsa_value()` also records the dimension of the bounding box.

### Examples

```
dbpath <- rbedrock_example_world("example1.mcworld")
db <- bedrockdb(dbpath)
```

```
# view all HSA in a world
hsa <- get_hsa_data(db, get_keys(db))
hsa
# add an HSA to a world
dat <- data.frame(x1 = 0, x2 = 15, z1 = 0, z2 = 15,
                  y1 = 40, y2 = 60, tag = "SwampHut")
put_hsa_data(db, dat, merge = TRUE)
close(db)
```

---

list_biomes	<i>List Minecraft Bedrock Edition biomes.</i>
-------------	---

---

### Description

List Minecraft Bedrock Edition biomes.

### Usage

```
list_biomes()
```

```
biome_id(x)
```

### Arguments

x	A character vector containing biome name.
---	---

---

locate_blocks	<i>Locate the coordinates of blocks in a chunk</i>
---------------	--

---

### Description

Locate the coordinates of blocks in a chunk

### Usage

```
locate_blocks(blocks, pattern, negate = FALSE)
```

### Arguments

blocks	A character array containing block data.
pattern	The pattern to look for. Passed to <code>stringr::str_detect</code> .
negate	If TRUE, return non-matching elements.

**Examples**

```
dbpath <- rbedrock_example_world("example1.mcworld")
db <- bedrockdb(dbpath)
blocks <- get_chunk_blocks_value(db, x=37, z=10, dimension=0)
locate_blocks(blocks, "ore")
close(db)
```

---

 minecraft\_worlds

*Utilities for working with Minecraft world folders.*


---

**Description**

`world_dir_path()` returns the path to the `minecraftWorlds` directory. Use options(`rbedrock.worlds_dir_path = "custom/path"`) to customize the path as needed.

`list_worlds()` returns a `data.frame()` containing information about Minecraft saved games.

`create_world()` creates a new Minecraft world.

`export_world()` exports a world to an archive file.

**Usage**

```
worlds_dir_path(force_default = FALSE)
```

```
list_worlds(worlds_dir = worlds_dir_path())
```

```
create_world(id = NULL, ..., worlds_dir = worlds_dir_path())
```

```
export_world(id, file, worlds_dir = worlds_dir_path(), replace = FALSE)
```

```
import_world(file, id = NULL, ..., worlds_dir = worlds_dir_path())
```

```
get_world_path(id, worlds_dir = worlds_dir_path())
```

**Arguments**

<code>force_default</code>	If TRUE, return most likely world path on the system.
<code>worlds_dir</code>	The path of a <code>minecraftWorlds</code> directory.
<code>id</code>	The path to a world folder. If the path is not absolute or does not exist, it is assumed to be the base name of a world folder in <code>worlds_dir</code> . For <code>import_world()</code> , if <code>id</code> is NULL a unique world id will be generated. How it is generated is controlled by the <code>rbedrock.rand_world_id</code> global options. Possible values are "pretty" and "mcpe".
<code>...</code>	Arguments to customize <code>level.dat</code> settings. Supports dynamic dots via <code>rclang::list2()</code> .
<code>file</code>	The path to an <code>mcworld</code> file. If exporting, it will be created. If importing, it will be extracted.
<code>replace</code>	If TRUE, overwrite an existing file if necessary.

## Examples

```
## Not run:  
  
create_world(LevelName = "My World", RandomSeed = 10)  
  
## End(Not run)
```

---

nbt\_byte

*Create an NBT value*

---

## Description

The Named Binary Tag (NBT) format is used by Minecraft for various data types. An NBT value holds a 'payload' of data and a 'tag' indicating the type of data held.

nbt\_\*(*x*) family of functions create nbt data types. unnbt(*x*) recursively strips NBT metadata from an NBT value.

payload(*x*) reads an nbt value's payload.

get\_nbt\_tag(*x*) returns the NBT tag corresponding to and NBT value.

## Usage

nbt\_byte(*x*)

nbt\_short(*x*)

nbt\_int(*x*)

nbt\_long(*x*)

nbt\_float(*x*)

nbt\_double(*x*)

nbt\_byte\_array(*x*)

nbt\_string(*x*)

nbt\_raw\_string(*x*)

nbt\_int\_array(*x*)

nbt\_long\_array(*x*)

nbt\_compound(...)

nbt\_list(...)

```
is_nbt(x)
payload(x)
unnbt(x)
get_nbt_tag(x)
```

### Arguments

x	An nbt value
...	Arguments to collect into an NBT compound or NBT list value. Supports dynamic dots via <code>rlang::list2()</code> .

---

PendingTicks

*Load and store PendingTicks NBT data*

---

### Description

PendingTicks data (tag 51) holds a list of NBT values for pending ticks.

`get_pending_ticks_data()` and `get_pending_ticks_value()` load PendingTicks data from db.

`get_pending_ticks_data()` will silently drop and keys not representing PendingTicks data. `get_pending_ticks_value()` supports loading only a single value. `get_pending_ticks_values()` is a synonym for `get_pending_ticks_data()`.

`put_pending_ticks_values()`, `put_pending_ticks_value()`, and `put_pending_ticks_data()` store PendingTicks data into db.

### Usage

```
get_pending_ticks_data(db, x, z, dimension)
get_pending_ticks_values(db, x, z, dimension)
get_pending_ticks_value(db, x, z, dimension)
put_pending_ticks_values(db, x, z, dimension, values)
put_pending_ticks_value(db, x, z, dimension, value)
put_pending_ticks_data(db, data)
```

### Arguments

db	A bedrockdb object.
x, z, dimension	Chunk coordinates to extract data from. x can also be a character vector of db keys.

values	A list of nbt objects
value	An nbt object.
data	A named-list specifying key-value pairs.

**Value**

get\_pending\_ticks\_data() returns a named-list of nbt data. get\_pending\_ticks\_values() returns a single nbt value.

---

put_values	<i>Write values to a bedrockdb.</i>
------------	-------------------------------------

---

**Description**

Write values to a bedrockdb.

**Usage**

```
put_values(db, keys, values, writeoptions = NULL)
```

```
put_value(db, key, value, writeoptions = NULL)
```

```
put_data(db, data, writeoptions = NULL)
```

**Arguments**

db	A bedrockdb object
keys	A character vector of keys.
values	A list of raw values.
writeoptions	A bedrock_leveldb_writeoptions object
key	A key that will be used to store data.
value	A raw vector that contains the information to be written.
data	A named-list of raw values, specifying key-value pairs.

**Value**

An invisible copy of db.

---

 RandomTicks

*Load and store RandomTicks NBT data*


---

### Description

RandomTicks data (tag 59) holds a list of NBT values for random ticks.

`get_random_ticks_data()` and `get_random_ticks_value()` load RandomTicks data from db.

`get_random_ticks_data()` will silently drop and keys not representing RandomTicks data. `get_random_ticks_value()` supports loading only a single value. `get_random_ticks_values()` is a synonym for `get_random_ticks_data()`.

`put_random_ticks_values()`, `put_random_ticks_value()`, and `put_random_ticks_data()` store RandomTicks data into db.

### Usage

```
get_random_ticks_data(db, x, z, dimension)
```

```
get_random_ticks_values(db, x, z, dimension)
```

```
get_random_ticks_value(db, x, z, dimension)
```

```
put_random_ticks_values(db, x, z, dimension, values)
```

```
put_random_ticks_value(db, x, z, dimension, value)
```

```
put_random_ticks_data(db, data)
```

### Arguments

<code>db</code>	A bedrockdb object.
<code>x, z, dimension</code>	Chunk coordinates to extract data from. <code>x</code> can also be a character vector of db keys.
<code>values</code>	A list of nbt objects
<code>value</code>	An nbt object.
<code>data</code>	A named-list specifying key-value pairs.

### Value

`get_random_ticks_data()` returns a named-list of nbt data. `get_random_ticks_values()` returns a single nbt value.

---

rbedrock_example	<i>Get path to rbedrock example</i>
------------------	-------------------------------------

---

### Description

rbedrock comes bundled with a number of sample files in its `inst/extdata` directory. This function make them easy to access.

### Usage

```
rbedrock_example(path = NULL)
```

```
rbedrock_example_world(path)
```

### Arguments

`path`                    Name of file or directory. If NULL, the examples will be listed.

### Examples

```
rbedrock_example()
rbedrock_example("example1.mcworld")
rbedrock_example_world("example1.mcworld")
```

---

read_level.dat	<i>Read and write data from a world's level.dat file.</i>
----------------	---

---

### Description

Read and write data from a world's level.dat file.

### Usage

```
read_level.dat(path, old = FALSE)
```

```
write_level.dat(object, path, old = FALSE, version = 8L)
```

### Arguments

`path`                    The path to a world folder. If the path does not exist, it is assumed to be the base name of a world folder in the local `minecraftWorlds` directory.

`old`                    Read/write to 'level.dat\_old' instead.

`object`                NBT data to be written to level.dat.

`version`                The level.dat format version for the file header.



**Value**

read\_level.dat returns nbt data.

write\_level.dat returns a copy of the data written.

**Examples**

```
# Fix level.dat after opening a world in creative.
dbpath <- rbedrock_example_world("example1.mcworld")
dat <- read_level.dat(dbpath)
dat$hasBeenLoadedInCreative <- FALSE
write_level.dat(dat, dbpath)
```

---

simulation_area	<i>Calculate a player-based simulation area</i>
-----------------	---

---

**Description**

Calculate a player-based simulation area

**Usage**

```
simulation_area(sim_distance, x = 0, z = 0)
```

**Arguments**

sim\_distance    A sim distance setting

x, z            Chunk coordinates where a player is standing

**Value**

A data.frame containing the chunk coordinates in the simulation area.

---

spawning_area	<i>Calculate a player-based spawning area</i>
---------------	---

---

**Description**

Calculate a player-based spawning area

**Usage**

```
spawning_area(sim_distance, x = 0, z = 0)
```

**Arguments**

sim\_distance     A sim distance setting  
 x, z             Chunk coordinates where a player is standing (can be fractional)

**Value**

A data.frame containing the chunk coordinates in the spawning area.

---

SubchunkBlocks             *Load and store SubchunkBlocks data*

---

**Description**

SubchunkBlocks data (tag 47) holds information about the blocks in a subchunks. Each chunk is divided into multiple 16x16x16 subchunks, and each subchunk is stored separately and indicated by the use of the subtag. Blocks are stored in a palette-based format. Subchunks can have two layers of blocks, and the extra layer is most-often used to store water for water-logged blocks.

These functions return block data as strings containing the block name and block states. The strings' format is blockname@state1=value1@state2=value2 etc. Blocks may have 0 or more states.

get\_subchunk\_blocks\_data() loads SubchunkBlocks data from a bedrockdb. It will silently drop and keys not representing SubchunkBlocks data. get\_subchunk\_blocks\_values() is a synonym for get\_subchunk\_blocks\_data().

get\_subchunk\_blocks\_value() loads SubchunkBlocks data from a bedrockdb. It only supports loading a single value.

get\_subchunk\_blocks\_from\_chunk() loads SubchunkBlocks data from a bedrockdb. It supports efficiently loading subchunk block data from a single chunk.

put\_subchunk\_blocks\_data(), put\_subchunk\_blocks\_values(), and put\_subchunk\_blocks\_value() store SubchunkBlocks data into a bedrockdb.

read\_subchunk\_blocks\_value() decodes binary SubchunkBlock data.

subchunk\_origins() returns a matrix containing the block coordinate of the lower NW corner of subchunk keys

subchunk\_coords() determines the block coordinates of blocks based on their array indexes and their subchunk origins.

**Usage**

```
get_subchunk_blocks_data(
  db,
  x,
  z,
  dimension,
  subchunk,
  names_only = FALSE,
  extra_block = !names_only
```

```
)

get_subchunk_blocks_values(
  db,
  x,
  z,
  dimension,
  subchunk,
  names_only = FALSE,
  extra_block = !names_only
)

get_subchunk_blocks_value(
  db,
  x,
  z,
  dimension,
  subchunk,
  names_only = FALSE,
  extra_block = !names_only
)

get_subchunk_blocks_from_chunk(
  db,
  x,
  z,
  dimension,
  names_only = FALSE,
  extra_block = !names_only
)

put_subchunk_blocks_data(db, data, version = 9L)

put_subchunk_blocks_values(db, x, z, dimension, subchunk, values, version = 9L)

put_subchunk_blocks_value(db, x, z, dimension, subchunk, value, version = 9L)

read_subchunk_blocks_value(
  rawdata,
  missing_offset = NA,
  names_only = FALSE,
  extra_block = !names_only
)

write_subchunk_blocks_value(object, version = 9L, missing_offset = NA_integer_)

subchunk_origins(keys)
```

```
subchunk_coords(ind, origins = subchunk_origins(names(ind)))
```

### Arguments

<code>db</code>	A bedrockdb object.
<code>x, z, dimension</code>	Chunk coordinates to extract data from. <code>x</code> can also be a character vector of db keys.
<code>subchunk</code>	Subchunk indexes to extract data from.
<code>names_only</code>	A logical scalar. Return only the names of the blocks, ignoring block states.
<code>extra_block</code>	A logical scalar. Append the extra block layer to the output (separated by ";"). This is mostly useful if you have waterlogged blocks. If the extra block is air, it will not be appended.
<code>data</code>	A named list of 16x16x16 character() arrays
<code>version</code>	Which format of subchunk data to use
<code>values</code>	A list of 16x16x16 character() arrays
<code>value</code>	A 16x16x16 character array
<code>rawdata</code>	a raw vector holding binary SubchunkBlock data
<code>missing_offset</code>	subchunk offset to use if one is not found in rawdata
<code>object</code>	A 16x16x16 character array.
<code>keys</code>	A character vector of database keys.
<code>ind</code>	Numeric vector or a named list of numeric vectors containing indexes for blocks in a subchunk.
<code>origins</code>	A matrix of subchunk origins.

### Details

If a subchunk contains only air it will not be stored in the database, and missing subchunks are considered air.

### Value

`get_subchunk_blocks_data()` returns a list of the of the values returned by `read_subchunk_blocks_value()`.

`get_subchunk_blocks_value()` and `read_subchunk_blocks_value()` return a 16x16x16 character array. The axes represent the x, y, and z dimensions in that order.

`get_subchunk_blocks_from_chunk()` returns a list of the of the values returned by `read_subchunk_blocks_value()`.

`read_subchunk_blocks_value()` returns a 16x16x16 character array. The axes represent the x, y, and z dimensions in that order.

`subchunk_coords()` returns a 3-column matrix of block coordinates.

# Index

ActorDigest, [2](#), [3](#), [4](#)  
Actors, [3](#), [3](#)

bedrock\_random, [6](#)  
bedrock\_random\_create\_seed, [7](#)  
bedrock\_random\_get\_double  
    (bedrock\_random), [6](#)  
bedrock\_random\_get\_float  
    (bedrock\_random), [6](#)  
bedrock\_random\_get\_int  
    (bedrock\_random), [6](#)  
bedrock\_random\_get\_uint  
    (bedrock\_random), [6](#)  
bedrock\_random\_seed (bedrock\_random), [6](#)  
bedrock\_random\_state (bedrock\_random), [6](#)  
bedrockdb, [4](#)  
biome\_id (list\_biomes), [26](#)  
Biomes, [8](#)  
BlockEntity, [9](#)

Checksums, [10](#)  
chunk\_blocks, [12](#)  
chunk\_blocks<- (chunk\_blocks), [12](#)  
chunk\_keys, [13](#)  
chunk\_origin, [14](#)  
chunk\_origin<- (chunk\_origin), [14](#)  
chunk\_origins (chunk\_keys), [13](#)  
chunk\_positions (chunk\_keys), [13](#)  
chunk\_tag\_int (chunk\_keys), [13](#)  
chunk\_tag\_str (chunk\_keys), [13](#)  
ChunkVersion, [11](#)  
close.bedrockdb (bedrockdb), [4](#)  
create\_acdig\_keys (ActorDigest), [2](#)  
create\_chunk\_keys (chunk\_keys), [13](#)  
create\_world (minecraft\_worlds), [27](#)

Data2D, [14](#)  
Data3D, [16](#)  
delete\_values, [17](#)  
Entity, [3](#), [4](#), [18](#)

export\_world (minecraft\_worlds), [27](#)

FinalizedState, [19](#)

get\_acdig\_data (ActorDigest), [2](#)  
get\_acdig\_value (ActorDigest), [2](#)  
get\_actors\_data (Actors), [3](#)  
get\_actors\_value (Actors), [3](#)  
get\_biomes\_data (Biomes), [8](#)  
get\_biomes\_value (Biomes), [8](#)  
get\_biomes\_values (Biomes), [8](#)  
get\_block\_entity\_data (BlockEntity), [9](#)  
get\_block\_entity\_value (BlockEntity), [9](#)  
get\_block\_entity\_values (BlockEntity), [9](#)  
get\_checksums\_data (Checksums), [10](#)  
get\_checksums\_value (Checksums), [10](#)  
get\_checksums\_values (Checksums), [10](#)  
get\_chunk\_blocks\_data, [20](#)  
get\_chunk\_blocks\_value  
    (get\_chunk\_blocks\_data), [20](#)  
get\_chunk\_blocks\_values  
    (get\_chunk\_blocks\_data), [20](#)  
get\_chunk\_version\_data (ChunkVersion),  
    [11](#)  
get\_chunk\_version\_value (ChunkVersion),  
    [11](#)  
get\_chunk\_version\_values  
    (ChunkVersion), [11](#)  
get\_data (get\_values), [24](#)  
get\_data2d\_data (Data2D), [14](#)  
get\_data2d\_value (Data2D), [14](#)  
get\_data2d\_values (Data2D), [14](#)  
get\_data3d\_data (Data3D), [16](#)  
get\_data3d\_value (Data3D), [16](#)  
get\_data3d\_values (Data3D), [16](#)  
get\_entity\_data (Entity), [18](#)  
get\_entity\_value (Entity), [18](#)  
get\_entity\_values (Entity), [18](#)  
get\_finalized\_state\_data  
    (FinalizedState), [19](#)

- get\_finalized\_state\_value  
(FinalizedState), 19
- get\_finalized\_state\_values  
(FinalizedState), 19
- get\_hsa\_data (HSA), 24
- get\_hsa\_value (HSA), 24
- get\_hsa\_values (HSA), 24
- get\_keys, 22
- get\_legacy\_biomes\_data (Biomes), 8
- get\_legacy\_biomes\_value (Biomes), 8
- get\_legacy\_biomes\_values (Biomes), 8
- get\_nbt\_data, 22
- get\_nbt\_tag (nbt\_byte), 28
- get\_nbt\_value (get\_nbt\_data), 22
- get\_nbt\_values (get\_nbt\_data), 22
- get\_pending\_ticks\_data (PendingTicks),  
29
- get\_pending\_ticks\_value (PendingTicks),  
29
- get\_pending\_ticks\_values  
(PendingTicks), 29
- get\_random\_ticks\_data (RandomTicks), 31
- get\_random\_ticks\_value (RandomTicks), 31
- get\_random\_ticks\_values (RandomTicks),  
31
- get\_subchunk\_blocks\_data  
(SubchunkBlocks), 34
- get\_subchunk\_blocks\_from\_chunk  
(SubchunkBlocks), 34
- get\_subchunk\_blocks\_value  
(SubchunkBlocks), 34
- get\_subchunk\_blocks\_values  
(SubchunkBlocks), 34
- get\_value (get\_values), 24
- get\_values, 24
- get\_world\_path (minecraft\_worlds), 27
  
- has\_values (get\_values), 24
- HSA, 24
  
- import\_world (minecraft\_worlds), 27
- is\_bedrockdb (bedrockdb), 4
- is\_nbt (nbt\_byte), 28
  
- list\_biomes, 26
- list\_worlds (minecraft\_worlds), 27
- locate\_blocks, 26
  
- minecraft\_worlds, 27
  
- nbt\_byte, 28
- nbt\_byte\_array (nbt\_byte), 28
- nbt\_compound (nbt\_byte), 28
- nbt\_double (nbt\_byte), 28
- nbt\_float (nbt\_byte), 28
- nbt\_int (nbt\_byte), 28
- nbt\_int\_array (nbt\_byte), 28
- nbt\_list (nbt\_byte), 28
- nbt\_long (nbt\_byte), 28
- nbt\_long\_array (nbt\_byte), 28
- nbt\_raw\_string (nbt\_byte), 28
- nbt\_short (nbt\_byte), 28
- nbt\_string (nbt\_byte), 28
  
- parse\_chunk\_keys (chunk\_keys), 13
- payload (nbt\_byte), 28
- PendingTicks, 29
- put\_acdig\_data (ActorDigest), 2
- put\_acdig\_value (ActorDigest), 2
- put\_actors\_data (Actors), 3
- put\_actors\_value (Actors), 3
- put\_biomes\_data (Biomes), 8
- put\_biomes\_value (Biomes), 8
- put\_biomes\_values (Biomes), 8
- put\_block\_entity\_data (BlockEntity), 9
- put\_block\_entity\_value (BlockEntity), 9
- put\_block\_entity\_values (BlockEntity), 9
- put\_chunk\_blocks\_data  
(get\_chunk\_blocks\_data), 20
- put\_chunk\_blocks\_value  
(get\_chunk\_blocks\_data), 20
- put\_chunk\_blocks\_values  
(get\_chunk\_blocks\_data), 20
- put\_chunk\_version\_data (ChunkVersion),  
11
- put\_chunk\_version\_value (ChunkVersion),  
11
- put\_chunk\_version\_values  
(ChunkVersion), 11
- put\_data (put\_values), 30
- put\_data2d\_data (Data2D), 14
- put\_data2d\_value (Data2D), 14
- put\_data2d\_values (Data2D), 14
- put\_data3d\_data (Data3D), 16
- put\_data3d\_value (Data3D), 16
- put\_data3d\_values (Data3D), 16
- put\_entity\_data (Entity), 18
- put\_entity\_value (Entity), 18
- put\_entity\_values (Entity), 18

put\_finalized\_state\_data  
    (FinalizedState), 19  
put\_finalized\_state\_value  
    (FinalizedState), 19  
put\_finalized\_state\_values  
    (FinalizedState), 19  
put\_hsa\_data (HSA), 24  
put\_hsa\_value (HSA), 24  
put\_hsa\_values (HSA), 24  
put\_legacy\_biomes\_data (Biomes), 8  
put\_legacy\_biomes\_value (Biomes), 8  
put\_legacy\_biomes\_values (Biomes), 8  
put\_nbt\_data (get\_nbt\_data), 22  
put\_nbt\_value (get\_nbt\_data), 22  
put\_nbt\_values (get\_nbt\_data), 22  
put\_pending\_ticks\_data (PendingTicks),  
    29  
put\_pending\_ticks\_value (PendingTicks),  
    29  
put\_pending\_ticks\_values  
    (PendingTicks), 29  
put\_random\_ticks\_data (RandomTicks), 31  
put\_random\_ticks\_value (RandomTicks), 31  
put\_random\_ticks\_values (RandomTicks),  
    31  
put\_subchunk\_blocks\_data  
    (SubchunkBlocks), 34  
put\_subchunk\_blocks\_value  
    (SubchunkBlocks), 34  
put\_subchunk\_blocks\_values  
    (SubchunkBlocks), 34  
put\_value (put\_values), 30  
put\_values, 30  
  
RandomTicks, 31  
rbedrock\_example, 32  
rbedrock\_example\_world  
    (rbedrock\_example), 32  
read\_acdig\_value (ActorDigest), 2  
read\_checksums\_value (Checksums), 10  
read\_chunk\_version\_value  
    (ChunkVersion), 11  
read\_data2d\_value (Data2D), 14  
read\_data3d\_value (Data3D), 16  
read\_finalized\_state\_value  
    (FinalizedState), 19  
read\_hsa\_value (HSA), 24  
read\_level.dat, 32  
read\_nbt (get\_nbt\_data), 22  
read\_nbt\_data (get\_nbt\_data), 22  
read\_subchunk\_blocks\_value  
    (SubchunkBlocks), 34  
  
simulation\_area, 33  
spawning\_area, 33  
subchunk\_coords (SubchunkBlocks), 34  
subchunk\_origins (SubchunkBlocks), 34  
SubchunkBlocks, 34  
  
unnbt (nbt\_byte), 28  
update\_checksums\_data (Checksums), 10  
  
worlds\_dir\_path (minecraft\_worlds), 27  
write\_acdig\_value (ActorDigest), 2  
write\_checksums\_value (Checksums), 10  
write\_chunk\_version\_value  
    (ChunkVersion), 11  
write\_data2d\_value (Data2D), 14  
write\_data3d\_value (Data3D), 16  
write\_finalized\_state\_value  
    (FinalizedState), 19  
write\_hsa\_value (HSA), 24  
write\_level.dat (read\_level.dat), 32  
write\_nbt (get\_nbt\_data), 22  
write\_nbt\_data (get\_nbt\_data), 22  
write\_subchunk\_blocks\_value  
    (SubchunkBlocks), 34