# MRF User Guide

For MRF Version 1.x

# MRF: Definition, Context and Introduction

MRF, short for Meta Raster Format, is a technology that combines raster storage with cloud computing. While the main target domain is cloud GIS, the MRF technology could also be used in other areas, such as medical imaging and scientific data processing.

MRF is:

* A raster storage format
* A tile cache format for web services
* A dynamic tile cache for another raster

For the purpose of this document, a raster is defined as a two dimensional array of values. In information technology, a raster most commonly represents an image, in which case the array elements are known as pixels, short for picture elements. An image can be grayscale or color. In the latter case multiple values are associated with each pixel, usually one for each of the red, green and blue components. In scientific applications, rasters are commonly used to represent sampled scalar fields or matrices, where each array value is the numeric value of the scalar field at a specific point. In geographic information science (GIS), a raster can be either a map image or an array of values. A raster is a very compact and efficient way of storing uniformly sampled data, since the set of coordinates does not have to be stored with each of the data values. Instead, the coordinates are calculated by knowing the raster projection, one or more reference points and resolution.

There are many raster formats in use today. Most of them have started as image formats, usually intended for disk storage or archiving. GIF, JPEG, PNG and TIFF are some of the best known examples. As image formats, they usually support grayscale and color images of reasonable size, and employ various compression algorithms for reducing the amount of storage needed. Most of these formats have been designed before the advent of the internet, and have continued to be used since they serve their purpose very well. Yet these formats have significant limitations, for example when used to store extremely large images, or non-image data.

MRF was designed to leverage existing image formats while addressing some of their shortcomings. In the simplest form, MRF explicitly provides tiling, indexing and multiple resolutions (aka overviews, pyramid, or resolution-sets) support. This is an extremely common approach, especially in GIS, allowing localized areas to be read without having to read the complete raster. It also allows for raster sizes well beyond what is achievable by traditional image formats. Since the tiles in an MRF file may themselves be stored in a raster format, the MRF is suitable as the tile storage format for web services. MRF can also be used as a cloud raster cache format, to improve the performance of web applications. MRF segregates the data, index and metadata in different files, allowing different classes of storage to be used for the different components as needed, enhancing efficiency, even on a single system.

There are of course other technologies that try to address the same areas. For example the naive approach of leaving the image tiles in folders and imposing a known folder and file naming strategy. This has the advantage that no special tools or applications are needed to explore and curate the larger dataset. This approach is somewhat fragile and does not scale as well as it seems at first glance, since the file and operating system overhead is increasing significantly with dataset size. A slightly better approach is to use a database for storing the tiles. In general a database has less overhead and thus scales better than the file in folder approach. The disadvantage is that a database engine is needed while most of the database functionality (tables, queries, transactions) are not useful or applicable to raster tile storage. In addition, databases expect and are optimized for smaller records than the normal raster tile size. The two dimensional grid intrinsic to a raster is not a common database construct, and tools for populating a raster database are scarce, non-standard or have to be written from scratch.

MRF takes the middle road between these two approaches. It provides scalability by providing only the needed database functionality parts. It does not rely heavily on the file system for tile management. It acts as a raster itself, so it can be read and written using raster aware applications. Performance and scalability have been the main design goals for the MRF, closely followed by usability and flexibility. The MRF is implemented as a GDAL driver (Geospatial Data Abstraction Layer), which allows the MRF to be immediately leveraged in many GIS applications, and providing access to well documented tools and workflows. As with most technologies, understanding the features and limitations by MRF is important if good results are to be expected.

# MRF File Structure

An MRF dataset has three components, metadata, index and data. Usually each component is stored in a separate file, but alternative configurations can be used.

* The metadata file represents the raster itself. It is an XML formatted file, which improves readability and extensibility. In GDAL, the XML content can be used directly instead of a file name, so the metadata file does not even need to exist as such. The metadata file uses the .mrf extension by convention, but any other file extension can also be used.
* The index file is concerned with the two dimensional organization of the raster tiles on a grid. It contains a two dimensional array of structures, each structure holding the size and offset of a tile. The index file size is proportional to the number of tiles that can reside in an MRF. The organization of the index file depends on which MRF features are being used, but for a single raster they are stored in a top-left aligned, row major array. The index file name is by default the same as the metadata file name with the .idx extension replacing the original .mrf.
* The data file contains the raster tiles forming the MRF, which themselves contain the data values for each pixel. There is no implicit order of the tiles in the data file. The datafile is modified only by appending at the end of the file, all existing content will continue to take space on disk, even if it is no longer accessible by the MRF driver.

Note that neither the index, nor the data file contain any information about the MRF size, organization or content. All three files are required for accessing the MRF content, it is usually not possible to recover the data from one or two of the files.

# Referring to an MRF

The normal reference to a specific MRF raster is to use to the metadata file name. The metadata file can have any extension, the file format detection in GDAL is done by matching the first ten characters in the file, which have the value “<MRF\_META>”. For example this command will work if the test.mrf is an MRF metadata file.

gdalinfo test.mrf

Another way to reference an MRF is to use the XML content of the metadata file as a reference. In this case the data and index file names have to be explicitly set in the metadata, since it is not possible to derive them based on the metadata file name. When this method is used from the command line, shell special characters have to be escaped, so that a correctly formed XML string is passed to the GDAL open command.

# MRF Operations

When a MRF is created, all three files are usually created on disk, in the same folder. One of the MRF format features is that the content can be read as soon as the files are created, even before any data is actually written. It is also possible to read from an MRF file while it is being written into. Regions of an MRF that have not been written to automatically return the NoData value if the NoData is defined, or zero otherwise. This is also true for the overviews, they can be read as soon as the MRF file is flagged as having overviews.

## Overview Generation

An MRF can have either no overviews or all overviews, until all the raster fits into a single tile. There is no information in the MRF about individual overviews, so it is not possible to only have a selected few! If the overviews are not populated, they will return no-data or black.

The MRF driver contains code to generate internal overviews using bilinear interpolation, for overview scale of powers of 2. The GDAL overview generation code can also be used, in which case overviews at different scale factor (3, 4 …) can also be generated. Same rule applies; all levels have to exist until all raster fits in one tile. If the MRF isn’t already marked as having overviews, the scale between overviews is the first value passed to gdaladdo utility. So the first overview to be generated and populated has to be the largest one. It is also mandatory to generate all necessary overviews in sequence, since they are generated from the previous one. Usually the list of levels passed to gdaladdo should be all the needed powers of the scale factor, like this:

gdaladdo Test.mrf 2 4 8 16 32 64 128

Or for powers of 3:

gdaladdo Test.mrf 3 9 27 81

For convenience, unnecessary levels (the big values) will generate a warning but not a fault. It is not possible to change MRF overviews from one scale factor to another. It is however possible to generate the overviews multiple times.

## Overview sampling:

The MRF driver contains its own resampling code, based on averaging. The internal code has less overhead than the GDAL averaging and is usually faster. Use *–r avg* as the sampling option to gdaladdo with 2 as a scale factor to select this algorithm. Only scale 2 works correctly with this option! The MRF built-in sampler pads to the right and bottom of the image when needed. The normal GDAL sampler stretches the input as needed by repeating rows and/or columns. Both samplers do take the NoData into account. For the internal sampler, each band in averaged independently, band interpretation has no significance. GDAL resampling does take the band interpretation into account, if an alpha band exists and the opacity is zero for any pixel, GDAL will zero out all the other bands for that pixel. The internal sampler does not use a progress indicator.

Note that GDAL up to version 1.11 uses an incorrect step when generating overviews. This bug results in inefficient execution, larger than necessary file sizes and sometimes visible artifacts. This problem has been addressed and should not affect future versions of GDAL. Also, Use *–r average* to use the GDAL bilinear interpolation. The results differ slightly from the MRF internal sampler, due to the different padding. GDAL pads when necessary by duplicating pixel rows, in the middle of the image. The progress indicator is per generated level.

Use *–r nearest* (or no –r option), to use GDAL NearNb sampling. The progress indicator will be per generated level.

GDAL resampling takes into consideration both the noDataValue and the alpha band, setting to zero pixels where the alpha band is zero. To force gdal to ignore the alpha, set the create option PHOTOMETRIC=MULTISPECTRAL. This will set the photometric interpretation of all bands to unknown. The MRF –avg method is not subject to this behavior, it will keep the the data values even if the alpha band is zero.

In case of an MRF file with overviews, it is possible to open a single specific overview level. The overviews are identified by their numeral and not by the relative scale, with 0 being the largest overview. The syntax used for this is <filename>:MRF:L<n>

For example, this command will explicitly open the first overview level:

gdalinfo test.mrf:MRF:L0

## Inserting data in MRF

Using an MRF specific utility, mrf\_insert, it is possible to modify a part of an MRF and regenerate only the affected portions of the overviews. This facility makes it possible to build very large datasets efficiently, operating on small areas at a time.

This functionality relies on the internal MRF resampling (-avg described above), so it will only work for averaging mode and powers of two between levels.

Set create option APPEND\_SUBDATASET to true avoid deleting the MRF header file.

Since a Caching or Cloning MRF may be used at the same time by different processes, the MRF driver contains code that allows it to be written by multiple processes safely. This feature might be useful for other types of MRF, for example when mrf\_insert is used to update different areas of the same file, or when multiple third dimension MRF Z-Slices are written to at the same time. To turn on this feature, manually add a boolean attribute called mp\_safe with the value ***on*** to the Raster node of the MRF metadata. It is not on by default since it slows down the write operations somewhat. This feature has only been tested on Windows and Linux, and it depends on the specific operating and file system implementation, there might be configurations in which this operation fails.

# Types of tile compressions supported by MRF

Tiles in an MRF are stored using one of the supported packing or compression formats. Some of the formats are themselves standard raster formats like JPEG, TIFF and PNG, while others are only compression formats. The choice of the tile format is passed to the MRF driver using the GDAL create option called COMPRESS.

## RAW Compression

As the name suggest, the RAW format directly stores the tile array, in a row major order. PIXEL and BAND interleave are supported, as well as all the data types. The RAW format has no other options or features, and all the basic MRF functionality applies. If a NoData value is defined per band, tiles which have only NoData values in them are not stored on disk. If the NoData value is not defined, zero is used internal to the MRF. The MRF does not guarantee a specific order of the tiles in the data file.

## DEFLATE Compression

DEFLATE is a generic compression algorithm, implemented in the open source zlib library. In MRF it can be used in two ways, as a stand-alone tile packing mechanism and also as a second packing step to other compression formats. The second meaning is activated by adding DEFLATE:on to the free form list OPTIONS. Raw compression with the DEFLATE option is equivalent to the DEFLATE compression format, even though the content of the metadata file is different. These two commands should generate MRFs with identical size data files, although the tile order may differ.

gdal\_translate –of MRF –co COMPRESS=RAW -co OPTIONS=”DEFLATE:on” input.tif raw\_and\_deflate.mrf

gdal\_translate –of MRF –co COMPRESS=DEFLATE input.tif deflate.mrf

The zlib compression level is calculated from the QUALITY setting as level = floor(Quality/10). The default is 8, which is very good compression albeit slow. A quality setting of 60 is recommended as a tradeoff between compression speed and size. Level zero, corresponding to quality values under 10, means no compression.

The DEFLATE compression will generate zlib compatible tile headers by default. Gzip or no headers can be used instead, by setting the free form options GZ and RAWZ. If both are set, the headers will be gzip. The zlib header is 6 bytes and includes a fast checksum algorithm. The gzip header is slightly larger and uses a CRC32 as a checksum, only slightly slower than the zlib one. Raw deflate does not have a checksum nor a header and is slightly faster than either the gzip or zip. The speed difference is insignificant for most useful quality settings.

The following command will generate an MRF in which every tile is a gzip stream:

gdal\_translate –of MRF –co COMPRESS=RAW -co OPTIONS=”DEFLATE:on GZ:on” input.tif gzipped.mrf

Which is equivalent to:

gdal\_translate –of MRF –co COMPRESS=DEFLATE -co OPTIONS=”GZ:on” input.tif gzipped.mrf

The strategy used for compression by zlib can also be controlled. It only affects the algorithm used during the compression. The generated stream can always be decompressed. For exact details on what the strategy flags refer to the zlib documentation. The free form option is Z\_STRATEGY, and the valid values are:

Z\_FILTERED: Skips the optional filtering of the input stream

Z\_HUFFMAN\_ONLY: Only the Huffman encoding part of DEFLATE is performed

Z\_RLE: Somewhat like an RLE, within the limits of DEFLATE

Z\_FIXED: Fixed Huffman tables

The compression speed and the size of the output will change significantly if these options are used.

## PNG and PPNG Compression

PNG is a lossless compression image format which uses the DEFLATE algorithm internally. PNG is the default compression mechanism for MRF. PNG compression is used for both the PNG and PPNG compression mode. PPNG stands for Palette PNG. While both types can have a MRF level palette, PPNG also stores the palette inside each and every PNG tile. Normally this should only be used if the PNGs are to be served over the web as color PNGs, otherwise the PNG compression results in smaller data files.

The PNG format itself supports up to sixteen bit unsigned integer data types. In the MRF, only the eight and sixteen bit formats are used. However, the MRF itself can have a signed sixteen bit data type (Int16), in which case the 16bit unsigned values stored in the PNG are interpreted as signed.

The QUALITY setting controls the DEFLATE stage of the PNG, with the same meaning as the ones described in the DEFLATE compression. Similarly, the Z\_STRATEGY band option controls the zlib stage of PNG. Choosing Z\_RLE or Z\_HUFFMAN\_ONLY will result in much faster compression, at the expense of size, Z\_HUFFMAN\_ONLY being the fastest. Z\_FIXED and Z\_FILTERED have much less effect. The effect of the strategy setting is much stronger than the QUALITY value setting.

Example of gdal\_translate options for PNG

-of MRF –co COMPRESS=PNG –co OPTIONS=”Z\_STRATEGY=Z\_RLE” –co QUALITY=50

## JPEG Compression

The JPEG compression depends on the internal GDAL libjpeg. It can handle 8 or 12 bit data. It can have up to 10 bands in pixel interleave mode. Note that only 8 bit JPEGs with 1 or 3 bands are suitable for web services in most cases.

The QUALITY setting is directly passed to JPEG library as the Q factor, the default value being 85. Values between 0 and 100 are supported, the common range being between sixty and eighty five, larger values producing visually better results at the cost of increased size. For the exact interpretation of Q, please consult JPEG documentation. For three bands interleaved, a couple of encoding options are available, controlled via the PHOTOMETRIC setting. The default setting should be used most of the time. Tiles produced this way are read correctly by most applications:

* DEFAULT: JPEG using YCbCr, 4:2:0 sampling. This provides good compression and visual quality.
* YCC: Compressed as YCbCr, 4:4:4 sampling, ie not spatially resampled. This setting produces files about a third larger than the default, which has fewer spatial and color artifacts. The color conversion itself results in significant loss of information.
* RGB: Compressed as RGB, not color converted and not spatially resampled. This setting produces much larger JPEG files. Files are about three times larger than with the default setting. MRF with this setting can be decoded and re-encoded multiple times at the same quality without any data quality degradation.

Optimizing the Huffman encoding tables for each tile, as opposed to using the default value can be enabled by having the “OPTIMIZE=ON” in the OPTIONS list. Choosing this will increase encoding time and reduce the tile size slightly, both are relatively small changes in most cases.

The 12 bit JPEG is used when the input data type is Int16 or UInt16.

## JPNG Compression

The JPNG compression chooses between JPEG and PNG encoding per tile, based on the presence of opacity. It is intended to be used as a web cached, to be rendered in the browser.

## LERC Compression

Limited Error Raster Compression (LERC) is an Esri compression format. The main benefit of using LERC is extremely fast compression and decompression when compared with PNG and even JPEG, as well as excellent compression for data types larger than eight bit. LERC is a single band compression, with an explicit NoData mask. This means that for MRF with LERC compression only band interleave is supported. LERC also supports a datamask, which in MRF is enabled when the NoData value is defined. The LERC built in NoData support makes it a great choice for storing sparse data.

LERC can be either lossy or lossless. LERC maximum error value (LERC\_PREC) is a floating point number that controls the quantization of the input data, thus the accuracy of the data. LERC may modify the input values but the change is always less or equal to the LERC maximum error value. The quanta or precision of the output data values will thus be twice the LERC\_PREC value. If the LERC maximum error is zero or too small for any space savings to be obtained by quantization, the input data values are not modified, and LERC becomes a lossless compression format. There are two versions of LERC compression supported in MRF, LERC and LERC V2 (default). LERC supports integer and floating point data types with up to 24 bits of precision. LERC V2 supports more data types with higher precision and is somewhat faster. LERC V2 also includes different compression methods that sometimes results in significantly better compression than LERC. Yet for most cases, the compression achieved will be very similar.

For integer types the default LERC\_PREC value is 0.5, corresponding to lossless compression. For floating point types the LERC\_PREC defaults to 0.001 (.002 value resolution). The compression achieved by LERC heavily depends on the LERC\_PREC value, which should be carefully selected for each particular dataset.

To set a custom LERC precision value, use the free form MRF OPTIONS mechanism, the option name being “OPTIONS”. To set the LERC precision for a new MRF, use the create option like this:

-co OPTIONS="LERC\_PREC=0.005"

To use LERC instead of the default LERC2, add V1=ON to the options string, like this:

-co OPTIONS="LERC\_PREC=0.01 V1=ON"

MRF tiles compressed with LERC can be further encoded with zlib (DEFLATE), which sometimes results in better compression at a slight expense of speed. DEFLATE speed is asymmetric, decompression being faster than compression, so it does not affect read speeds as much as it does writes. To add DEFLATE to LERC, add “DEFLATE:ON” to the list of options. This example sets both the LERC precision and the extra DEFLATE option:

-co OPTIONS="LERC\_PREC=0.01 DEFLATE=ON"

Once set, the LERC\_PREC value will be used for all subsequent writes into the respective MRF.

## TIFF Compression

In the TIFF compression, every tile is a TIFF raster which uses the lossless LZW internal compression. Most data types are supported. Note that the tiles are not GeoTiffs, they do not contain geotags.

# Types of MRF

## Static MRF

This is the name for the basic storage format MRF, where all the three files are physically sitting in the same folder. In use it is similar to a TIFF or many other raster formats.

## Split MRF

The three files that compose an MRF (metadata, index and data) can be distributed across different storage systems. This is accomplished by having two extra XML nodes in the MRF metadata file, each containing GDAL accessible file names for the index or respectively for the data file. These types of files are not created by gdal\_translate and need to be created by manually editing the metadata file. The two nodes to be added are <IndexFile> and <DataFile>. They are added as sub-nodes of the <Raster> node. The content is simply a path to where the data or the index file can be found. The Split MRF can be used for example to accelerate access, by keeping the metadata files and possibly the index file on a faster storage (local SSD) while having the large data files on a HDD or a NAS. Other than the file location, there is no difference between the Static and the Split MRF. The IndexFile and DataFile nodes can also contain an optional attribute called ***offset***, with a numerical value. This value will be added to the normal, calculated file offsets for all access to the respective files. This feature can be exploited to combine the data and index files of an MRF.

## Caching MRF

MRF can also be used as an intermediary format, to cache data another raster file. The original raster is called the ***source*** raster, while the MRF used to cache becomes the ***caching MRF***. Only reading from a caching MRF is exposed in GDAL, writing to the MRF files occurs automatically. Opening a caching MRF for update is not supported. It is also not possible to write to the parent dataset through a caching MRF. Some of the GDAL functionality of the parent raster might not be available when accessing the data through an MRF. Only access to the raster data, the geotransform and projection are guaranteed to be available. Only static rasters, including static/split MRFs should be cached. Chaining caching MRFs is possible but cache coherency may become an issue.

### Creating a MRF cache

#### CACHEDSOURCE Create Option

The MRF GDAL driver only supports the CreateCopy, so the simplest way to instantiate a caching MRF is using the gdal\_translate. In addition to the normal MRF create options, the creation of a caching MRF dataset requires the presence of the “**CACHEDSOURCE**” option, whose value is the file name of the raster dataset to be cached. Only local files are supported, in any format readable by GDAL. The file name should be absolute, except for the case where the parent raster file is located in the same exact folder as the caching MRF metadata file.

An example of creating a caching MRF:

gdal\_translate –of MRF –co **CACHEDSOURCE= H12003\_MB\_1m\_MLLW\_14of16.tif** \data\LERC\_test\H12003\_MB\_1m\_MLLW\_14of16.tif \data\LERC\_test\tst.mrf

In the command above, the presence of the CACHEDSOURCE option flags the file as a caching MRF and gets stored in the MRF metadata file. The value is the file name without the absolute path, which means that the caching mrf metadata file will always reside in the same location as the parent dataset file.

The command above will create the caching MRF metadata, data and index files and proceed to copy the parent dataset into the caching MRF. The caching MRF has the same structure as a normal, static MRF, except that it is flagged as a caching MRF. This example is not a true MRF caching application, since all the data is copied into the new created file at creation time. It is possible to erase the index and data files and then use the MRF for caching, the index and data of a caching MRF file are created when needed. WARNING: Always remove the index and data files together, otherwise errors will occur.

#### NOCOPY Create Option

To initialize a caching MRF but not store any data in it, use the Boolean create option **NOCOPY**=**True**. For example:

gdal\_translate -of MRF -co COMPRESS=LERC -co BLOCKSIZE=512 -co OPTIONS="LERC\_PREC=0.01" -co **NOCOPY=True** -co CACHEDSOURCE=H12003\_MB\_1m\_MLLW\_14of16.tif \data\LERC\_test\H12003\_MB\_1m\_MLLW\_14of16.tif \data\LERC\_test\tst.mrf

The combined use of the CACHEDSOURCE and the NOCOPY options should be the most common use. Normally, the source raster as used on the gdal\_translate command line and the value of the CACHEDSOURCE are identical. The source raster is used as the source of data and metadata during the gdal\_translate execution. The CACHEDSOURCE raster is used later, when reading from the caching MRF, if the data is not present. While this syntax seems clumsy, it is required due to the structure of gdal\_translate, and it also offers the possibility to initialize a caching MRF using a local file while caching a different, possibly remote raster.

The example above, in addition to the precedent one, sets the caching MRF compression to LERC, sets the blocksize to be used, sets the LERC max error via the freeform option and sets the NOCOPY to true. This will leave the caching MRF initialized but empty. When raster blocks are then read from the MRF, data is read from the CACHEDSOURCE raster and stored in the caching MRF. On subsequent reads, if the data already exists in the caching MRF it is no longer read from the parent dataset.

#### UNIFORM\_ SCALE Create Option

The MRF (caching or normal) can be created with the full set of internal overlays. This is especially useful when creating a caching MRF.

gdal\_translate -of MRF -co COMPRESS=LERC -co BLOCKSIZE=512 -co OPTIONS="LERC\_PREC:0.01" -co **UNIFORM\_ SCALE=2** -co NOCOPY=True -co CACHEDSOURCE=H12003\_MB\_1m\_MLLW\_14of16.tif \data\LERC\_test\H12003\_MB\_1m\_MLLW\_14of16.tif \data\LERC\_test\tst.mrf

Note that the overlays will be written with data read at the corresponding scale from the parent dataset, thus they might be different from the ones that could be created via gdaladdo command. However, gdaladdo will still work on the caching MRF, especially if the base level is fully populated already. Exercise this option with caution, the current implementation of the overlay building reads the destination tile before writing to it. In the case of a caching MRF, this will result in fetching the tile from the parent source, storing a copy in the caching MRF, then reading the higher resolution source from the MRF, scaling by 2 and writing the tile again into the caching MRF.

#### Using a caching MRF

This is the easy part, simply use the caching MRF for reading data just as any other raster format in GDAL. When opened, the MRF driver will also open the source dataset. When reading, if the tile already exists in the caching MRF, then it will be read from it. Otherwise, the tile will be requested from the source and a copy stored in the caching MRF before returning it to the requestor. Thus, the first time a tile is requested it will have the source performance, any subsequent writes will have local performance.

#### Advanced use of caching MRF

The two extra features of a caching MRF over a static one, fetching content from a different source and storing content locally can be individually turned off. Turning them both off will transform the caching MRF into a static MRF, where only the content that already exists within the cache is accessible. The ability to turn these features off and then turn them on again is done via file access rights. The state of these features is set when the data and index files are opened, and they will persist for that task as long as those files are kept open.

Turning off the **local cache writes** while still requesting data from the source still allows reading the cached content as well as source content. It is useful for example when the local cache should no longer be allowed to increase in size. To turn off local cache writing off, make the existing MRF **data file** read only.

Turning off the **new content fetch** is useful for reading only the local cache, or when the source is no longer available. It avoids the latency and penalty of trying and failing to access the source. To turn off the source fetching, make the existing MRF **index file** read only. Turning off the new content fetch will implicitly turn off local cache writes, since there is no new content to be written. If a caching MRF uses the same file for both data and index, this will be the behavior.

Sometimes it is useful to temporarily stop the caching MRF from storing data locally while preserving data access to the remote data source. This can be achieved by setting the environment variable **MRF\_BYPASSCACHING** to **TRUE**. This variable can also be set as a gdal configuration option. All caching and cloning MRF files opened while this variable is set to true are affected, it is not possible to selectively choose which caching MRFs are affected.

The performance of the caching MRF depends on a multitude of factors, including the page sizes of both the caching MRF and the remote files. Good performance is achieved when the caching MRF and the remote file have the same page size. A particular case is when the remote is pixel interleaved but the caching MRF is band interleaved (as in the case of LERC compression). In this case, the remote page may be read and decompressed multiple times, once for each and every output band. Only if the GDAL block cache is large enough to hold all the blocks this will not happen and the blocks will be reused. If the source page size is not efficient for the user application, it is recommended that the source data be reformatted ahead of time with a suitable page size, possibly as MRF.

## Cloning MRF

As a further optimization, if the source dataset of a caching MRF is itself an MRF, and caching MRF have the identical structure with the source one (image size, projection, page size, compression …), the caching MRF can eliminate the page transcoding, copying the already compressed pages from the source MRF. This type of MRF is called a Cloning MRF, since it is an almost identical copy of the source MRF. Creating a clone MRF cannot be done using gdal\_translate, since it is not possible to insure that the source has the same properties as the caching MRF. Instead, a cloning MRF should be created by copying the cloned MRF metadata file to where the cloning MRF should reside and adding the following lines to the top level node:

<CachedSource>

<Source clone=”true>/path/to/cloned.mrf</Source>

</CachedSource>

The data and the index files for a cloned MRF will be created on read, as needed. Only static or split MRFs can be cloned, the cloning MRF does not trigger the full GDAL block reads to the source dataset. This characteristic has the added benefit of reducing the GDAL block cache use, since the source blocks are not read in the block cache.

## Versioned MRF

A versioned MRF is a special type of static MRF. It has to be created by hand, adding the “versioned” Boolean attribute to the Raster node in the metadata file. Once set up as a versioned MRF, any tile overwrite will automatically create a new version within the same MRF set of files. There is no support for explicit version creation. Versions are counted from 0 (latest, default), 1 being the oldest, 2 the second oldest. Only the version 0 (latest) is being updated, all the other versions are read only. By using the normal file name as the MRF reference, the latest version is being used. To read from a previous version, use the ornate file name with a V prefix. This reference will open the oldest version of an versioned MRF

<filename>:MRF:V1

The number of versions is not limited. The index file will grow as versions are added. It is possible to have overviews in a versioned MRF. To make this work, create an empty MRF while reserving space for overviews, then modify the metadata to flag it as versioned. Alternatively, once the initial version of the base resolution has been created, overviews should be generated, and only then the MRF be flagged as being versioned.

## Third dimension MRF

A raster is usually a 2D dataset, characterized by the size in the X and Y dimensions and the number of color bands or channels. MRF supports an optional third dimension, dimension Z. An MRF with the Z size N contains N 2D rasters, all with the same size in X and Y, same number of bands, stored with the same exact parameters. Each of the N 2D rasters is identified by the Z index, an integer from 0 to N-1. In GDAL only one of the 2D rasters is available at a time, determined at the time of the file opening, with Z index zero being the default. In other words, a normal, 2D MRF is a 3D MRF with the Z size of 1. The Z dimension is not visible in GDAL, other than the ZSIZE and ZSLICE metadata items in the IMAGE\_STRUCTURE metadata domain.

While the third dimension MRF seems similar to the versioned MRF, it is structurally different, and there are other differences:

* Similar to the X and Y sizes, the Z size is a constant that is set when the MRF file is created and cannot be modified later. The number of versions is not limited and can change during use.
* Each 2D image in a 3rd dimension MRF can be written to, in any order. In a versioned MRF, only writes to the latest version can be done

#### Z Slice selection

To open a specific Z index from a 3rd dimensional MRF, the file name ornament Z has to be used. The syntax used for this is <filename>:MRF:Z<n>. If the Z index is not specified, zero is used.

For example, this command will explicitly open the 2D raster at Z index 1:

gdalinfo TenSlice.mrf:MRF:Z1

It is also possible to select a specific overview of a specific Z index, for example the second overview (scale 4 if powers of two are used) of the slice index five:

gdalinfo TenSlice.mrf:MRF:Z5:L1

#### Creating and writing to a 3rd dimension MRF

To create an MRF with the Z size different from one, the ZSIZE create option has to be set at creation time. This gdal\_translate command will create an MRF with the Z size of 10, getting the size and other parameters from the source.tif file and leaving the output MRF empty.

gdal\_translate –of MRF –co ZSIZE=10 –co NOCOPY=true source.tif TenSlice.mrf

If the NOCOPY parameter is not set, the data from source.tif will also be copied in the slice index zero of the TenSlice.mrf. The UNIFORM\_SCALE parameter can also be used, reserving space but not populating the overviews. Using the Z index selection names, gdaladdo can also be used to reserve the index space for the overviews and populate them, for any valid Z index value.

Once the 3rd dimension MRF exists, the full resolution image at a Z index different than one can be inserted using gdal\_translate. Note that the create options have to be identical with the ones used to initially create the MRF. Also, if the input images have to have the same exact size and most of the other parameters, except for the ones explicitly overwritten by the create options. For example, content from a JPEG and a DEFLATE geotiff can be copied into the same 3rd dimension MRF, but the MRF compression has to be explicitly set in the gdal\_translate command. If this condition is not met, the output MRF may be corrupted. This is because the MRF metadata file will be recreated by every gdal\_translate command and no tests are done to insure that the content is the same.

The following commands will copy a source3.tif into the z index 3 of the mrf created above, followed by populating the overviews for the slice:

gdal\_translate –of MRF –co ZSIZE=10 source3.tif TenSlice.mrf:MRF:Z3

gdaladdo –r avg TenSlice.mrf:MRF:Z3

## Lerc data MRF

The MRF driver also recognizes and reads a lerc compressed data-file, if it is one of the supported fromats. This type of data behaves as a read-only single tile MRF with LERC compression, without geo-reference. This feature is mostly intended to be used by the GDAL WMS driver. An open option, DATATYPE, can be used to change the data type when reading from LERC V1 compressed data. The default data type for LERC V1 is byte. LERC V2 can only be read as the same data type it was encoded as, the DATATYPE open option is ignored.

## Overwriting an MRF

When overwriting an MRF, GDAL normally tries to erase the files if they exist. To avoid having the data or the index file erased un-intentionally, the MRF driver does not do this. This means that if a file exists and is used repeatedly as a destination for gdal\_translate, the data file will keep growing and the index file will keep its old content, which is the desired behavior. This can create problems in certain cases, for example when the same file name is reused for images of different size or structure, or when the MRF itself is corrupt. Crashes may occur in some of these situations. In these cases, the index and data file should be erased by hand, outside of the GDAL infrastructure.

# APENDIX A, MRF Metadata Schema

# APENDIX B, Index file format

The MRF index is a vector of tile records. A tile index record is sixteen bytes long and contains the tile offset and size, each stored as an eight byte unsigned integer. In C, a tile index record is defined as

typedef struct {

uint16\_t offset;

uint16\_t size;

} tidx;

Example values in this document will be using the notation [Offset, Size] for a tile index. By convention the index for a tile which has no data written into it has the size of zero [R,0]. This index record will generate a tile filled with zeros or NoData on read. The value for the offset is reserved and should be written as zero. An offset value of 1 and a size of 0 [1,0] is used by the caching/cloning MRFs to signify that a tile is zero or NoData in the source file. It will be read as zero without triggering a read from the source.

The order of the tile records in the index file depends on the type of MRF. Tile records are usually stored in top-left orientation, in row major order (Y,X).

If the MRF contains multiple channels (Bands) and they are stored as band interleaved data, the band index changes first, then the spatial index (Y, X, C).

If the MRF Z dimension is more than 1, each Z slice index is stored consecutively, thus Z varies after Y (Z, Y, X, C).

If the MRF contains versions, the current version is stored at the start of the index file, immediately followed by version one and so on (V, Y, X, C) or (V, Z, Y, X, C).

If there are overviews, the tile index vectors for the overviews immediately follow the index vector for the full resolution, in the decreasing order of resolution, (l;X,Y,C) or (l;Z,X,Y,C). Note that the vector for an overview level is smaller than the vector for the previous overview or base resolution.

For cloning MRFs, the index of the local cache data is followed immediately by a copy of the cloned MRF index. The content of both may be updated during reads.

# APENDIX C, Create Options

In GDAL, a list of key-value string pairs can be used to pass various options to the target driver. Using the gdal\_translate utility, these options are passed using the –co Key=Value syntax. Most of the names of the options supported by MRF have been chosen to match the ones used by TIFF. The create options supported by MRF are:

|  |  |  |
| --- | --- | --- |
| Key | Default Value | Description |
| BLOCKSIZE | 512 | The tile size, in both X and Y |
| BLOCKXSIZE | 512 | Horizontal tile size |
| BLOCKYSIZE | 512 | Vertical tile size |
| COMPRESS | PNG | Choses the tile packing algorithm |
| ZSIZE | 1 | Specifies the third dimension size |
| INTERLEAVE | PIXEL or BAND, format dependent | PIXEL or BAND interleave |
| NETBYTEORDER | FALSE | If true, for some packings, forces endianness dependent input data to big endian when writing, and back to native when reading |
| QUALITY | 85 | An integer, 0 to 100, used to control the compression |
| PHOTOMETRIC |  | Sets the interpretation of the bands and controls some of the compression algorithms |
| SPACING | 0 | Reserve this many bytes before each tile data |
| NOCOPY | False | Create an empty MRF, do not copy input |
| UNIFORM\_SCALE |  | Flags the MRF as containing overviews, with a given numerical scale factor between successive overviews |
| CACHEDSOURCE |  | GDAL raster reference to be cached in the caching MRF being created |

# APENDIX D, Free-Form Create Options

In addition to the normal create options, MRF also supports a set of options that control features of only certain packing formats, or can be used to modify default behaviors. The main difference between the create options and free-form options is that the latter are saved in the MRF metadata file and may apply when a file is read, not only when it is written. The free form options are not part of the GDAL interface, and as such they are not checked for correctness when passed to the driver. If a free form option doesn’t seem to have the expected effect, the exact spelling should be checked, they are case sensitive.

The free-form option list is passed as a single string value for the create option called OPTIONS. The value is a free form string containing white space separated key value pairs. GDAL list parsing is being used when reading, either the equal sign “=” or the colon “:” can be used as the separator between key and value.

When using gdal\_translate utility, the free form option syntax will be:

-co OPTIONS=”Key1=Value1 Key2=Value2 …”

|  |  |  |  |
| --- | --- | --- | --- |
| Key | Default Value | Format | Description |
| DEFLATE | False | Most | Apply zlib DEFLATE as a final packing stage |
| GZ | False | DEFLATE | GZIP headers |
| RAWZ | False | DEFLATE | No headers |
| Z\_STRATEGY |  | PNG, DEFLATE | DEFLATE algorithmic choice  Z\_HUFFMAN\_ONLY, Z\_FILTERED, Z\_RLE, Z\_FIXED |
| V2 | False | LERC | Uses LERC V2 compression |
| LERC\_PREC | 0.5 for integer types  0.001 for floating point | LERC | Maximum value change allowed |
| OPTIMIZE | False | JPEG | Optimize the Huffman tables for each tile. Always true for JPEG12 |

# APENDIX E, Open Options

In GDAL 2.x API, a list of key-value string pairs can be used to pass various options to the target driver when reading. Using the gdal\_translate utility, these options are passed using the –oo Key=Value syntax.

|  |  |  |
| --- | --- | --- |
| Key | Default Value | Description |
| DATATYPE | Byte | Sets data type for reading raw LERC V1 files |