RasterLite

a very simple C library [*and related management tools*]
implementing an efficient storage solution for huge raster data sources based on
SQLite + **SpatiaLite** Spatial DBMS

Index of contents:

Introduction: refreshing some useful basic notions

Surely you already know all this: this one simply is a very quick and fast reminder, to better fix the operative context.

A very commonly used kind of **GIS** data source is represented by **raster** imagery: this identifies some kind of *georeferenced* digital image, i.e. explicitly stating how and where the image has to be placed over a conventional map. In other words, in a georeferenced raster image, each single pixel explicitly corresponds to some map coordinate.

There are two commonly used ways to georeference a raster image:

- using the canonical **GeoTIFF** format: this allows to embed any information belonging to the spatial reference system and georeferencing directly within the image itself.
- using an **ESRI world file**, i.e. a small text file adopting a well known format: this is by far less desirable, because the georeferencing infos are store into a separate file (*and not within the image itself*), and any spatial reference system explicit setting is unsupported.
- anyway, this one isn't a real issue, because the **geotifcp** tool allows you to get a true GeoTIFF from an ESRI world file in a very simple way.

Raster imagery foundations:

Technically speaking a raster image is a two-dimensional rectangular matrix of individual **pixels**. Each pixel corresponds (directly or indirectly) to some well defined **color**. And colors are defined as **RGB** values, i.e. as a triplet of values indicating the **Red**, **Green** and **Blue** relative intensity.
Most usually an **8-bit** color depth is used, and so a **0** value corresponds to "*completely off*", i.e. black, and **255** to "*completely on*", i.e. pure red or pure green or pure blue.
Any real color can be encoded using an appropriate RGB value: an 8-bit color depth allows RGB to represent **16,777,216** different colors, i.e. the so called **true color** commonly used by digital cameras, screens, color printers and other well known digital imagery equipments.
This is also known as the **RGB color space** (*commonly used on color digital photography*): each pixel requires 3 bytes, i.e. *24 bits* to be represented into this color space.

RGB hex value	Red	Green	Blue	Color	Example
`0x000000`	0	0	0	Black	⬛
`0xFFFFFF`	255	255	255	White	⬜
`0x808080`	128	128	128	Medium Gray	▒
`0xD0D0D0`	208	208	208	Light Gray	░
`0xFF0000`	255	0	0	Red	🟥
`0x00FF00`	0	255	0	Green	🟩
`0x0000FF`	0	0	255	Blue	🟦
`0xFFFF00`	255	255	0	Yellow	🟨
`0x00FFFF`	0	255	255	Cyan	🟦
`0xFF00FF`	255	0	255	Magenta	🟪

Please note: as you can easily notice from the above table, a full 256-levels gray scale can always be represented using a single channel, because any *gray* value always has identical values for Red, Green and Blue. And this defines the **GRAYSCALE color space** (*commonly used on black and white digital photography*): each pixel requires a single byte, i.e. *8 bits* to be represented into this color space.

In the **MONOCHROME color space** only two colors are supported (*usually, black and white*): each pixel requires a *single bit*, i.e. one single byte can store 8 pixel. If you are wondering about a common example for *monochrome* (aka *bi-level*) images, keep in your hands the latest fax you've received, or a page just printed from your black and white laser printer.

As an alternative way, we can define a **PALETTE-based color space**. The palette stores a limited set of RGB values (*usually, max. 256*), and consequently each pixel doesn't requires any longer a full RGB value: it will simply store a palette index, thus indirectly retrieving the corresponding RGB value: consequently, each pixel requires a single byte, i.e. *8 bits* to be represented into this color space.

Quite obviously, a lot of different color spaces exists: but they aren't too much widespread, so we can ignore them at all.

Color Space	Single Pixel Size	Notes
RGB	3 bytes 24 bits	True color 16 million colors
GRAYSCALE	1 byte 8 bits	256 levels gray scale
MONOCHROME	1 bit	Bi-level Black or white [*no half tones*]
PALETTE	1 byte 8 bits	256 colors palette

Compression algorithms foundations:

Raster imagery usually require strong amounts of storage. And consequently, using detailed optimal resolution GIS raster imagery representing some extended territory, typically requires very huge and really impressive amounts of storage.

In order to reduce the amount of **disk storage** required by raster imagery, several very specific and commonly used compression algorithms can be applied.

But first to try compressing your raster imagery, keep well in your mind that choosing the correct color space can help you avoiding to waste unnecessarily your precious and limited disk space.
Simply storing as RGB an image that can harmlessly be stored as GRAYSCALE or PALETTE will unusefully waste 3-times the actual size you really need.
And storing as RGB a MONOCHROME image will waste 24-times the actual size you really need.

<u>*Loseless compression:*</u>

A *loseless* compression represents a completely reversible operation. The result of the compression actually requires only a small fraction of the original space, but you can always get back the original image, with no loss of information at all.

A typical example for loseless compression is represented by the well known *zipfile* algorithm.
You can zip and then unzip a single file (*or a whole folder*) as many times you wish: you always will get the same identical file from where you started, with no difference at all.

<u>*Lossy compression:*</u>

A *lossy* compression represents an irreversible operation. You never can get back an uncompressed image exactly identical to the original one, because some kind of information suppression will be introduced anyway during the compression process.
Typically, *lossy* compression algorithms can squeeze your images more much better than *loseless* algorithms does, but at the cost of some irreversible information suppression (and quality degradation).

And *lossy* compression algorithms enables you to select a variable **compression factor.**
You can choose to apply a strong (*very aggressive*) compression, in order to minimize the required space, but at the expense of a very modest quality.
Or you can alternatively choose to apply a moderate (*not so aggressive*) compression, sacrificing some extra space, but obtaining a good or excellent quality (*i.e. not the slightest artifact will be perceived by the naked eye of an human observer*).

<u>Commonly used image formats:</u>

<u>**TIFF** (***T****agged* ***I****mage* ***F****ile* ***F****ormat*)</u>

This actually represents a quite complex family of different formats.
GeoTIFF represents a TIFF-based superset (*fully compatible*) allowing to store georeferencing infos directly into the image file itself.
The TIFF sub-formats most commonly used are:

- **RGB**: true color uncompressed images
- **GRAYSCALE**: 256 gray tones uncompressed images
- **MONOCHROME**: bi-level uncompressed images
  - images of this kind can be compressed in a very efficient way applying the **CCITT FAX-3** or **CCITT FAX-4** algorithms. Both them implement a *loseless* compression.

TIFF images tends to requires huge amounts of disk storage, but are quite widespread used (namely as GeoTIFFs) because they allow to fully preserve the original resolution and quality.

JPEG (*Joint Photographic Experts Group*)

This format always implements a *lossy* compression, based on **DCT** (***D**iscrete **C**osine **T**ransform*) and *Hufman's encoding*.
Compression factor is selectable as a quality factor, this ranging from **Q=90** (smoothly compressed, but showing excellent quality) and **Q=20** (strongly compressed, but showing infamous quality). Reasonably quality factors to be used can be:

- **Q=90**: very good quality, moderate compression
- **Q=75**: good quality, optimal compression
- **Q=60** or **Q=50**: moderate quality, strong compression
- any other setting (**Q < 50**) will produce very poor results

The JPEG supported sub-formats are:

- **RGB**: true color compressed (*lossy*) images; this is the standard format adopted by the digital photography market.
- **GRAYSCALE**: 256 gray tones compressed (*lossy*) images: this exactly corresponds to *old-fashioned* black and white pictures.

PNG (*Portable Network Graphics*)

This format always implements a *loseless* compression, based on *filtering* and *DEFLATE*, an algorithm very closely related to the well known *zip*.
The PNG commonly supported sub-formats are:

- **RGB**: true color compressed (*loseless*) images. please note: such a *loseless* compression will preserve the full original image quality, but cannot achieve the compression efficiency allowed by JPEG.
- **GRAYSCALE**: 256 gray tones compressed (*loseless*) images. please note: in this case too the original image quality will be fully preserved, but compression efficiency is by far worst than the one you can obtain using JPEG.
- **PALETTE**: 256 colors palette based, (*loseless*) compressed images.

GIF (*Graphics Interchange Format*)

This format was quite hated during the latest year, because the **LZW** (***L**empel-**Z**iv-**W**elch*) *loseless* compression algorithm it applies was patent covered: but in 2003-2004 any pending patent definitively expired.
The GIF only supports the **PALETTE** color space: i.e. 256 colors palette based, (*loseless*) compressed images.
Compression efficiency of GIF compared with PNG-PALETTE widely depends upon the image itself: sometimes PNG is better, but other times GIF may represent a best solution.

WAVELET compression

This format always implements a *lossy* compression, based on *Wavelet Transform*. There are several flavors of this algorithm, and many of them are patent covered, so they are completely unavailable for open source implementations.
rasterlite own implementation for WAVELET compression is based on the **epsilon** library originally developed by **Alexander Simakov**, <xander@entropyware.info>

Compression factor is selectable as a compression ratio, this ranging from **Q=25** (smoothly compressed, but showing excellent quality) and **Q=150** (strongly compressed, but showing infamous quality). Reasonably quality factors to be used can be:

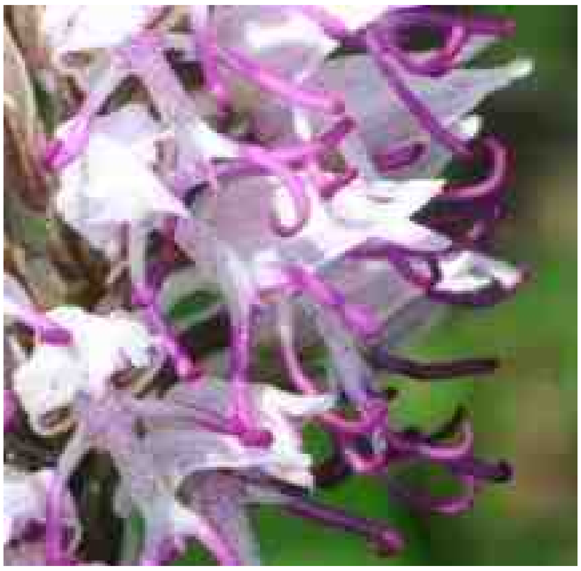
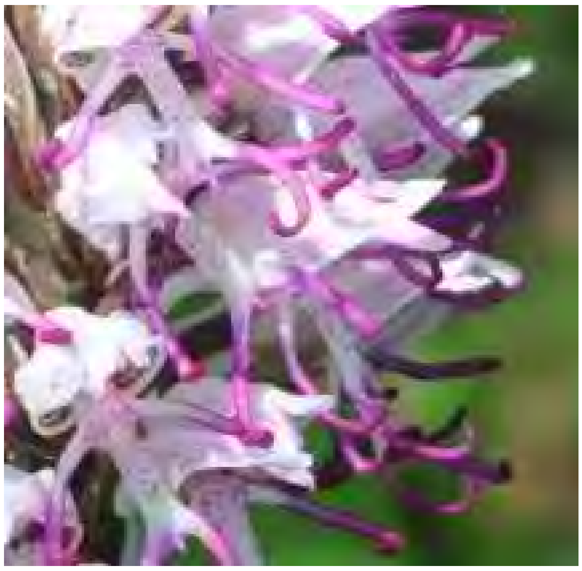
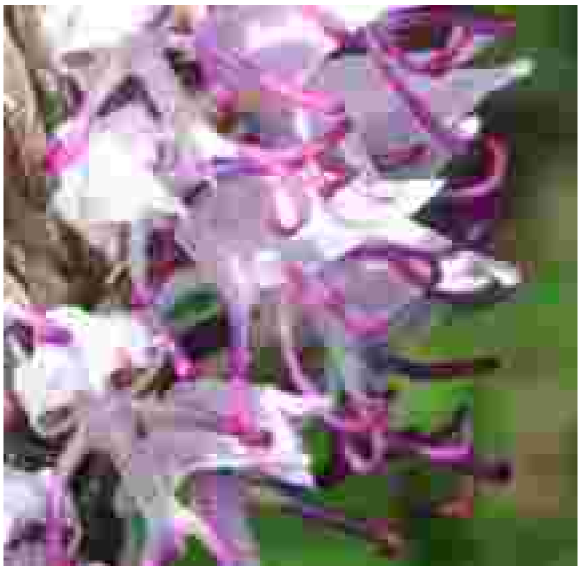
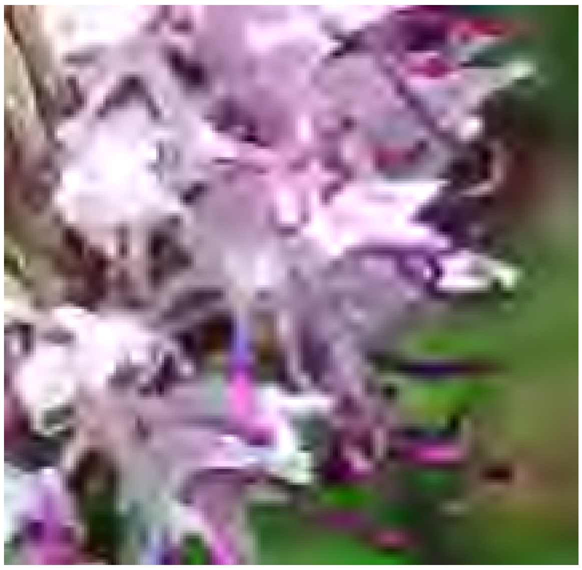
- **Q=25**: very good quality, moderate compression
- **Q=50**: good quality, optimal compression
- **Q=100**: moderate quality, strong compression
- any other setting (**Q > 100**) will produce very poor results

The WAVELET supported sub-formats are:

- **RGB**: true color compressed (*lossy*) images.
- **GRAYSCALE**: 256 gray tones compressed (*lossy*) images.

JPEG vs WAVELET

JPEG and WAVELET are conceptually similar algorithms, both supporting exactly the same color spaces and implementing a lossy compression.
But they are based on completely different mathematics, so they produce quite different visual effects.
As a rule of the thumb, at low compression factors they are quite identical: but at stronger compression factors JPEG is prone to show *big squared block* artifacts: WAVELET tends to show *blurry* artifacts. And WAVELET has the amazing capability to *squeeze* much more the compressed image, if you are so crazy to select some very high compression factor.

JPEG	WAVELET
 aggressive compression: you can easily notice big squared artifacts.	 aggressive compression: the image softens, but doesn't show any evident artifact
 very aggressive compression: the squared artifacts are really evident and annoying.	 very aggressive compression: the image is now too soft and shows a very noticeable *flou* effect

Color spaces and corresponding image formats

The following table shows allowable transformations between uncompressed TIFF and other compressed image formats:

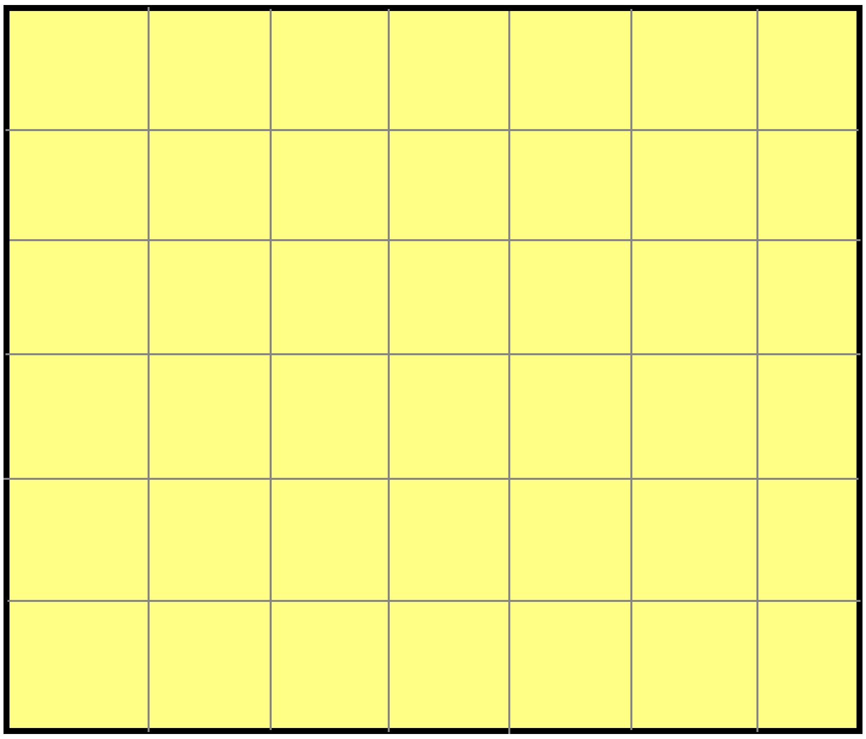
from original TIFF	to JPEG	to PNG	to GIF	to WAVELET
RGB uncompressed	RGB ***lossy*** compression optimal compression, good quality	RGB ***loseless*** compression not at all an astonishing compression, but quality is fully preserved	Not allowed will produce an unacceptable color space reduction	RGB ***lossy*** compression optimal compression, good quality
GRAYSCALE uncompressed	GRAYSCALE ***lossy*** compression optimal compression, good quality	GRAYSCALE ***loseless*** compression not at all an astonishing compression, but quality is fully preserved	PALETTE ***loseless*** compression ill-advised	GRAYSCALE ***lossy*** compression optimal compression, good quality
MONOCHROME uncompressed, or FAX-3 / FAX-4 ***loseless*** compressed	GRAYSCALE ***lossy*** compression ill-advised	GRAYSCALE ***loseless*** compression ill-advised	PALETTE ***loseless*** compression ill-advised	GRAYSCALE ***lossy*** compression ill-advised
PALETTE max. 256 colors uncompressed	RGB ***lossy*** compression ill-advised	PALETTE ***loseless*** compression good compression, quality is fully preserved	PALETTE ***loseless*** compression good compression, quality is fully preserved	RGB ***lossy*** compression ill-advised

Please note: compressed images require less disk space to be stored. But once you load them in memory, they will immediately be expanded to their original full size. And usually, a full RGB color space is used internally, so lots and lots of RAM are required in order to process raster imagery.

Tiling:

Raster imagery commonly used in GIS consist in really huge images: dimensions of 15,000 x 10,000 pixels are not at all uncommon: such a raster requires about 450 MB of memory, to be displayed over the screen. Handling so wide images implies several undesirable side effects:

- a lot of your precious RAM will be used. If your system merely has 512 MB of RAM, using so much RAM may produce a severe performance handicap.
- transferring a so huge amount of data from disk to RAM imposes an heavy load on the I/O sub-system: and I/O requires a long time to be completed, even if you are using a fast hard disk.
- If you try using some compression algorithm, you'll surely reduce the I/O load, and this may imply a strong benefit. But you are now imposing a very strong computational load on the CPU, because compression/decompression algorithms require a lot of calculation to be performed.



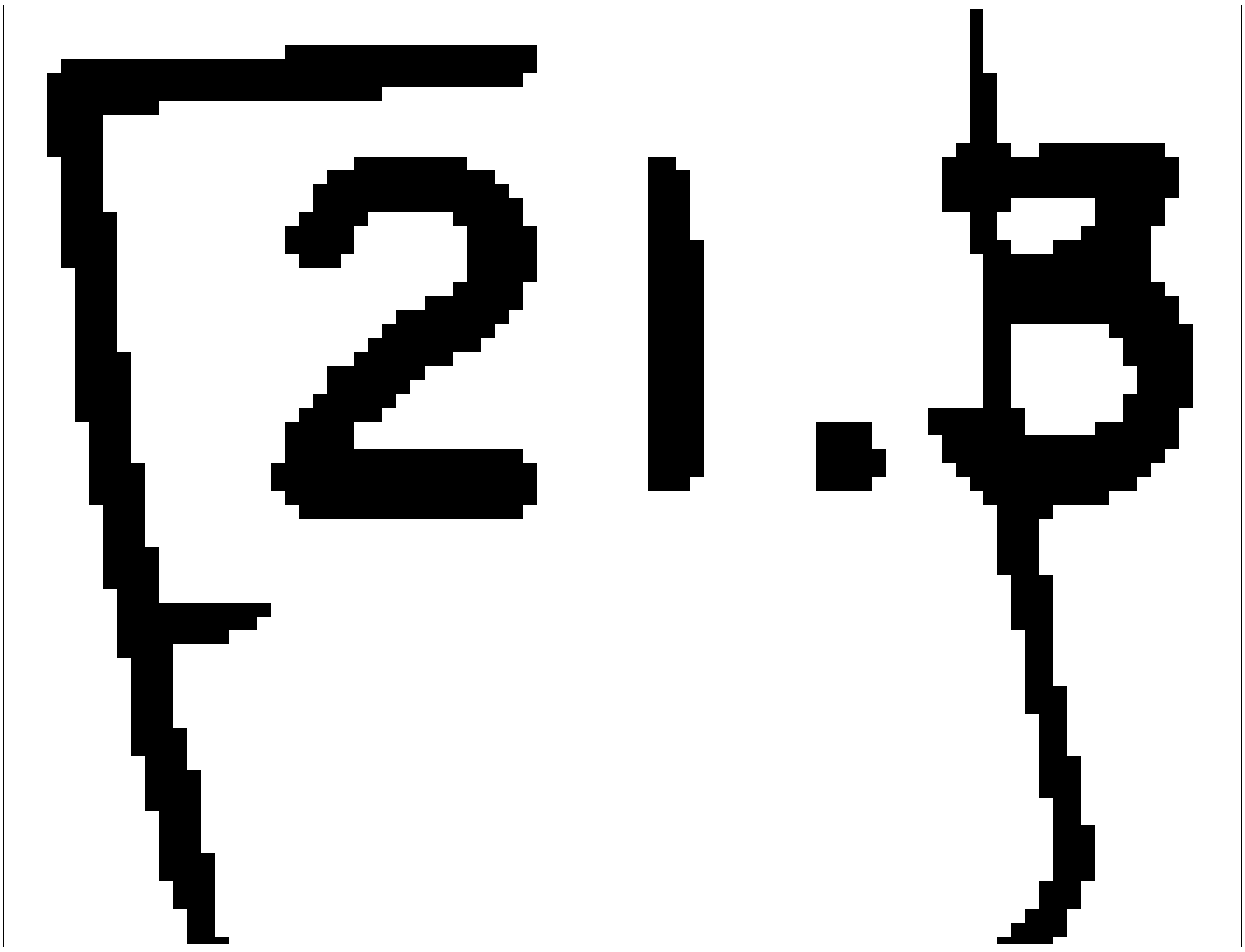
So, a very useful solution is the one to implement some kind of **tiling**. We can split the original huge image into several smaller images (*the individual tiles*). Now we can reconstruct anyway the full image (*simply reassembling together any required tile*), and we can now selectively load only the few tiles we really need at each time.

This one is a simple but effective way allowing to dramatically reduce the RAM requirements; and also has strong benefits in I/O terms. And decoding some small image requires few calculation, so the CPU overhead will not be so prohibitive anyway.

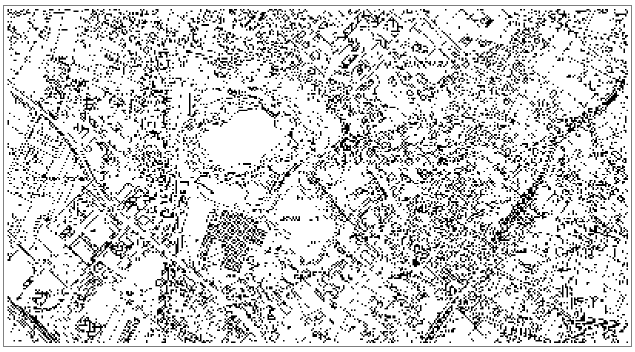
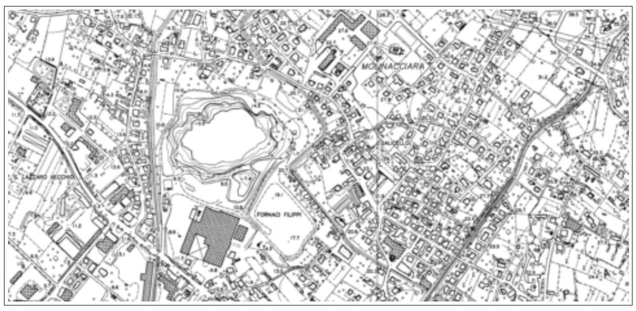
As from my own tests, using any modern CPU [Pentium4 and most recent ones] the benefits you can get by reducing the I/O load are by far biggest than the CPU overhead due to the task of decoding such compressed images.

Pyramids:

Raster images presents another hard-to-solve problem. Namely they can be displayed at one only scale, i.e. the one exactly corresponding to their original size.
And this may be really annoying, if you need to use them for some GIS application, freely using useful functions such as **zoom in** and **zoom out**, as ordinary vector-type GIS data sources allow with no problem at all.

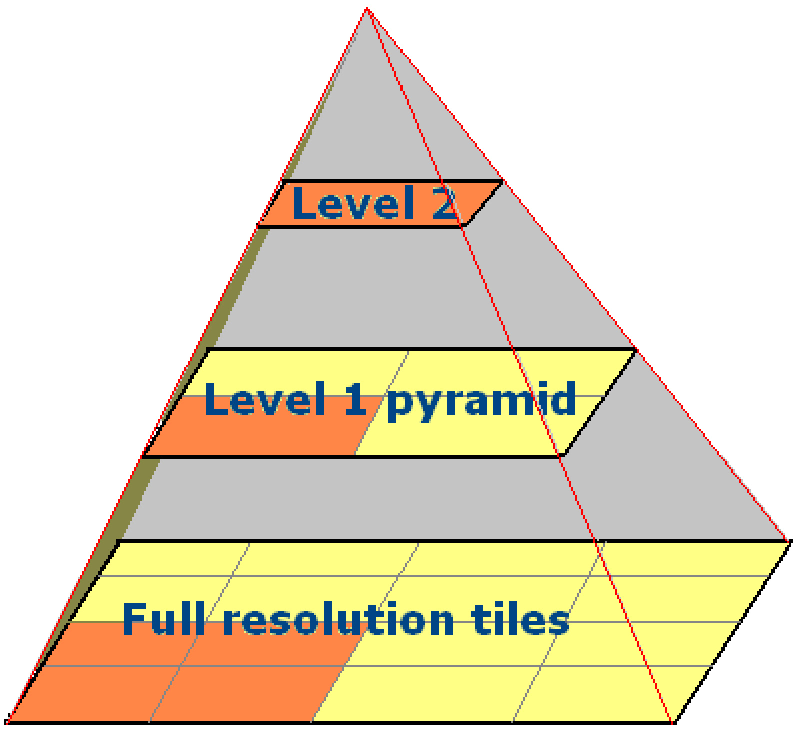


When you try to **zoom in** too much a raster-type data source, then individual pixels become absolutely evident as big solid squares. And this leads to a very poor visual effect.
Unhappily, there is very few you can do to avoid such ***pixelation*** artifact, because this require more information than the one actually available.

 Simple resizing	 Resizing combined with re-sampling

In the opposite direction, i.e. when you try to **zoom out** some raster image, you can get two quite different effects, depending on the algorithm you'll apply:

- you can apply a simple image resizing: this one is a very fast operation, but the result is visually very poor, as shown by the image on the left.
- alternatively, you can combine resizing and pixel re-sampling (*interpolation*); but such an operation requires an heavy computational load, i.e. is a very slow one to be performed.



In this case may be really helpful adopting a **pyramidal** structure, as follows:

- the lowermost level corresponds to the original tiles (full resolution, original one)
- any other level actually contains another tiled image, adopting a smallest dimension, and obtained applying re-sampling and interpolation. Such hi-quality reduced-size tiled image will be prepared only once, using an off-line process, and thus doesn't requires any further computation during run time.
- usually, each pyramid level corresponds to the immediate lowermost level, down sized by a factor 2.
- the topmost level simply contains a single small sized tile, representing the whole raster.

Merging together raster and Spatial DMBS technologies

A Spatial DBMS implements SQL support (*as any other plain DBMS does*), but supports Spatial Data (aka **Geometry**) as well.
Any Spatially enabled DBMS supports **R-Trees** (aka **Spatial Index**), thus supporting a fast way to retrieve data using Spatial relationships.
And finally, quite any DBMS supports a valid mechanism to store huge amounts of raster images, internally stored as **BLOB** (***B**inary **L**arge **Ob**ject*) data.

The **SpatiaLite** DMBS supports any of these basic Spatial DBMS capabilities, but has many more interesting features to be exploited:

- SpatiaLite simply is an extension based on the very popular and widespread **SQLite** DBMS: so it's extremely light-weighted, really simple to use, and doesn't require any kind of installation, configuration and optimization task.
- a complete SQLite + SpatiaLite DB is simply stored in a single file-system file. Such DB files implement a **cross-platform architecture**.
- so you can easily transfer a complete DB simply copying it; and you can safely perform such an operation even if you are using completely different and heterogeneous platforms.
- a SQLite DB isn't constrained by any dimensional upper limit; you can safely use a DB file requiring several GigaBytes, with no problem at all. And SQLite allows to store BLOBs in a really efficient way.

So, SpatiaLite really is a very good candidate to start with, when exploring advanced ad sophisticated integration between raster and Spatial DBMS technologies.
And this is exactly the goal pursued by the **RasterLite** library and related tools.

A **RasterLite** data source, i.e. a complete, and may well be very complex, raster data source, supporting **tiles** and **pyramids**, is based upon two correlated DB tables:

- the *tableprefix***_metadata** table: this is used to store individual tiles metadata.
- and the *tableprefix***_rasters** table: this is used to store individual tiles raster as BLOBs.
- there is a very good reason suggesting to split the **_metadata** and the **_rasters** tables: SQLite doesn't like too much fetching huge rows, as the ones storing BLOBs. So, adopting a separate table to store rasters will significatively reduce the I/O overhead, thus bringing to a noticeable performance bonus.
- and merging together the two tables when required isn't at all difficult. You simply have to perform a relational JOIN operation.

Table layout:

the *tableprefix*_**rasters** table layout:

Column	Data type	Clause	Notes
id	INTEGER	NOT NULL PRIMARY KEY AUTOINCREMENT	tile unique ID
raster	BLOB	NOT NULL	the tile raster image

the *tableprefix*_**metadata** table layout:

Column	Data type	Clause	Notes
id	INTEGER	NOT NULL PRIMARY KEY	tile unique ID *
source_name	TEXT	NOT NULL	the original source name i.e. the pathname identifying the file containing the original image.
tile_id	INTEGER	NOT NULL	progressive id identifying a tile within its own pyramid's level. **
width	INTEGER	NOT NULL	the tile horizontal dimension (pixels)
height	INTEGER	NOT NULL	the tile vertical dimension (pixels)
pixel_x_size	DOUBLE	NOT NULL	the horizontal dimension corresponding to a single pixel (expressed in map units). ***
pixel_y_size	DOUBLE	NOT NULL	the vertical dimension corresponding to a single pixel (expressed in map units). ***
geometry	POLYGON	NOT NULL	a Polygon corresponding to the tile MBR aka BBOX. (expressed in map units). Supported by an R-Tree Spatial Index

Notes:

*	a 1:1 relationship joins *tableprefix*_**rasters.id** and *tableprefix*_**metadata.id**
**	each pyramid's level restart numbering tiles from 0
***	all tiles belonging to the same pyramid's level must have exactly identical values for **pixel_x_size** and **pixel_y_size**

the **raster_pyramids** table layout:

Column	Data type	Clause	Notes
table_prefix	TEXT	NOT NULL	the **table_prefix**, i.e. the prefix to be prepended to both **_metadata** and **_rasters** complete table names
pixel_x_size	DOUBLE	NOT NULL	the horizontal dimension corresponding to a single pixel (expressed in map units).
pixel_y_size	DOUBLE	NOT NULL	the vertical dimension corresponding to a single pixel (expressed in map units).
tile_count	INTEGER	NOT NULL	the total number of tiles using this resolution

This latest simply is an utility table, supporting fast and quick identification of available pyramid's levels for a given raster data source.

Useful SQL queries:

The following SQL queries are exactly the ones used internally by the **RasterLite** library and related tools: you aren't supposed to never use them in a direct way, because they are managed internally by the library itself.
Anyway, they represent a very useful didactic material, helping to get a better in-depth understanding of actual RasterLite implementation.

```
SELECT pixel_x_size, pixel_y_size, tile_count
FROM raster_pyramids
WHERE table_prefix LIKE 'tableprefix'
ORDER BY pixel_x_size DESC
```

This SQL query retrieves all the available Pyramid's Levels for a given raster data source.

```
SELECT srid
FROM geometry_columns
WHERE f_table_name LIKE 'tableprefix_metadata'
      AND f_geometry_column LIKE 'geometry'
```

This SQL query retrieves the SRID [*Spatial* ***R****eference System* ***ID***] used by a given raster data source.

```
SELECT m.geometry, r.raster
FROM "tableprefix_metadata" AS m,
     "tableprefix_rasters" AS r
WHERE m.ROWID IN
    (
        SELECT pkid
        FROM "idx_tableprefix_metadata_geometry"
        WHERE xmin < frame_max_x
          AND xmax > frame_mix_x
          AND ymin < frame_max_y
          AND ymax > frame_min_y
    )
    AND m.pixel_x_size = x_size AND m.pixel_y_size = y_size
    AND r.id = m.id
```

This query fetches [*using the* ***R-Tree*** *Spatial Index*] any tiled raster image required in order to fill the required frame using the selected Pyramid's Level:

- ***frame_min_x***, ***frame_mix_y***, ***frame_max_x*** and ***frame_max_y*** identifies the MBR aka BBOX for the required frame, i.e. the map extent you wish to cover.
- ***x_size*** and ***y_size*** identifies the required Pyramid's Level.

Using the RasterLite management tools

The **RasterLite** library standard distribution includes three useful management tools, specifically aimed to help you while creating, feeding and testing a Raster Data Source.
All them are simple **CLI** (*Command Line Interface*) tools, so you have to launch them from the command shell specifying any required argument as appropriate.

Step 1: loading the original rasters into a Raster Data Source:

The ***rasterlite_load*** utility tool creates a new Raster Data Source (if not already exists), and then loads one or more **GeoTIFF** raster images into the data source, actually splitting the original image into individual tiles, and eventually compressing each tiled raster as required.

Step 2: creating Pyramid's Levels for a Raster Data Source:

The ***rasterlite_pyramid*** utility tool explores an already existing Raster Data Source generating any Pyramid's Level as required. You can safely execute many times ***rasterlite_pyramid***, because it works along the following guidelines:

- if any already existing Pyramid's Level is found, it will be completely deleted. Obviously, the original full-resolution tiles will be preserved anyway.
- then any required Pyramid's Level will be completely regenerated starting from the full-resolution tiles actually found at execution time.

Step 3: creating the TopMost Pyramid's Levels for a complex Raster Data Source:

The ***rasterlite_pyramid*** tool simply builds pyramids for each single GeoTIFF imported into the DBMS; so, if your Raster Data Source is a complex one (i.e. *using may individual GeoTIFFs*) you have to use the ***rasterlite_topmost*** utility tool in order to build the TopMost Pyramid's Levels tiles (i.e. *tiles joining more adjacent GeoTIFFs*). Creating such TopMost Level tile will speed up a lot visualizing the Raster Data Source as a whole. You can safely execute many times ***rasterlite_topmost***, because it works along the following guidelines:

- if any already existing TopMost Pyramid's Level is found, it will be completely deleted. Obviously, the original full-resolution tiles and the lowermost level tiles will be preserved anyway.
- then any required Pyramid's Level will be completely regenerated starting from the lowermost resolution tiles actually found at execution time.

Step 4: testing and checking a Raster Data Source:

The ***rasterlite_tool*** utility tool access an already existing Raster Data Source generating an arbitrary *frame* image as required. This is really useful by itself, and represents a powerful test tool as well.
You can export such *frame* images into the GeoTIFF, GIF, PNG or JPEG format, so you can easily use any ordinary *visual* SW in order to examine them.

As a general rule, RasterLite doesn't requires huge amounts of memory to work.
Even if you are processing really wide raster images [e.g. requiring some 500MB each one], you can safely try to process them using a standard PC with only 512MB RAM.
You'll be surprised when you'll check by yourself how low actually is the memory footprint required by RasterLite to run efficiently.

rasterlite_load **arguments:**

Short format	Long format	Expected value	Notes
`-?`	`--help`		Prints the arg-list and then exits
`-t`	`--test`		Performs any preliminary step, but doesn't alter at all the DB: useful for preliminary feasibility checking.
`-v`	`--verbose`		Verbose output [single tile progress]
`-d`	`--db_path`	SQLite + SpatiaLite DB path	Mandatory arg. The DB must exists, and must contain the SpatiaLite Metadata tables. [*spatial_ref_sys*, *geometry_columns*]
`-T`	`--table-name`	***tableprefix*** identifying the Raster Data Source	Mandatory arg. If such a data source doesn't exists, it will be created.
`-D`	`--dir-path`	a pathname identifying a directory containing one or more **GeoTIFF**s	Mandatory arg, mutually exclusive. You can use --`file` in order to load a single raster, or `–dir-path` in order to load any GeoTIFF found into the selected directory all-in-one.
`-f`	`--file`	a pathname identifying a single **GeoTIFF**	
`-s`	`--tile-size`	the preferred max. tile size	Optional arg, default = 512 You can freely select any integer value, but it will be impicitely reset into the range 128 - 8192
`-e`	`--epsg-code`	a valid EPSG SRID code	Optional arg. Usually any GeoTIFF includes its own EPSG code, but sometimes this is completely foolish or broken one. So you can use `--epsg-code` in order to forcibly override to some different soundest value.
`-i`	`--image-type`	the preferred image format to be used for individual tiles	Optional arg, default = JPEG Allowable values are: TIFF, PNG, GIF, JPEG and WAVELET *
`-q`	`--quality`	the adjustable quality setting for *lossy* compression algorithms supporting such an option	Optional arg, default = 75 for JPEG or = 25 for WAVELET Ignored in any other case.

* as explained in a previous paragraph, some constraints will restrict allowable image format conversions.

rasterlite_load applies the following schema to determine the actual image format to be used to store individual tiles:

GeoTIFF sub-format	Requested format	Format actually used
MONOCHROME	TIFF	TIFF MONOCHROME compressed as CCITT FAX-4 *loseless* compression
	PNG	
	GIF	
	JPEG	
	WAVELET	
GRAYSCALE	TIFF	TIFF GRAYSCALE uncompressed
	PNG	PNG GRAYSCALE *loseless* compression
	GIF	JPEG GRAYSCALE *lossy* compression
	JPEG	
	WAVELET	WAVELET GRAYSCALE *lossy* compression
PALETTE 256 colors	TIFF	TIFF PALETTE uncompressed
	PNG	PNG PALETTE *loseless* compression
	GIF	GIF *loseless* compression
	JPEG	
	WAVELET	
RGB	TIFF	TIFF RGB uncompressed
	PNG	PNG RGB *loseless* compression
	GIF	JPEG RGB *lossy* compression
	JPEG	
	WAVELET	WAVELET RGB *lossy* compression

rasterlite_pyramid **arguments:**

Short format	Long format	Expected value	Notes
`-?`	`--help`		Prints the arg-list and then exits
`-t`	`--test`		Performs any preliminary step, but doesn't alter at all the DB: useful for preliminary feasibility checking.
`-v`	`--verbose`		Verbose output [single tile progress]
`-d`	`--db_path`	SQLite + SpatiaLite DB path	Mandatory arg. The DB must exists, and must contain the SpatiaLite Metadata tables. [*spatial_ref_sys*, *geometry_columns*]
`-T`	`--table-name`	***tableprefix*** identifying the Raster Data Source	Mandatory arg.
`-i`	`--image-type`	the preferred image format to be used for individual tiles	Optional arg, default = PNG Allowable values are: TIFF, PNG, JPEG and WAVELET *
`-q`	`--quality`	the adjustable quality setting for *lossy* compression algorithms supporting such an option	Optional arg, default = 75 for JPEG or = 25 for WAVELET Ignored in any other case.

* Tiles belonging to any Pyramid's Level (*except the original full-resolution one*) must use the RGB color space anyway. This is because the re-sampling (*pixel interpolation*) algorithms used to generate the scaled-down images does apply *dithering*, thus effectively expanding the color space actually required.

rasterlite_topmost **arguments:**

Short format	Long format	Expected value	Notes
`-?`	`--help`		Prints the arg-list and then exits
`-t`	`--test`		Performs any preliminary step, but doesn't alter at all the DB: useful for preliminary feasibility checking.
`-v`	`--verbose`		Verbose output [single tile progress]
`-d`	`--db_path`	SQLite + SpatiaLite DB path	Mandatory arg. The DB must exists, and must contain the SpatiaLite Metadata tables. [*spatial_ref_sys*, *geometry_columns*]
`-T`	`--table-name`	***tableprefix*** identifying the Raster Data Source	Mandatory arg.
`-i`	`--image-type`	the preferred image format to be used for individual tiles	Optional arg, default = PNG Allowable values are: TIFF, PNG, JPEG and WAVELET *
`-q`	`--quality`	the adjustable quality setting for *lossy* compression algorithms supporting such an option	Optional arg, default = 75 for JPEG or = 25 for WAVELET Ignored in any other case.
`-c`	`--transparent-color`	hexadecimal RGB color 0xRRGGBB	Optional arg, default NONE
`-b`	`--background-color`	hexadecimal RGB color 0xRRGGBB	Optional arg, default BLACK (i.e. 0x000000)

* Tiles belonging to any Pyramid's Level (*except the original full-resolution one*) must use the RGB color space anyway. This is because the re-sampling (*pixel interpolation*) algorithms used to generate the scaled-down images does apply *dithering*, thus effectively expanding the color space actually required.

** *Please note*: you have to run `rasterlite_topmost` after running `rasterlite_pyramid`: and always remember, using `rasterlite_topmost` on a Raster Data Source containing only a single GeoTIFF make absolutely no sense.

***rasterlite_tool* arguments:**

Short format	Long format	Expected value	Notes
`-?`	`--help`		Prints the arg-list and then exits
`-o`	`--output`	the output image path	Mandatory arg.
`-d`	`--db_path`	SQLite + SpatiaLite DB path	Mandatory arg. The DB must exists, and must contain the SpatiaLite Metadata tables. [*spatial_ref_sys*, *geometry_columns*]
`-T`	`--table-name`	***tableprefix*** identifying the Raster Data Source	Mandatory arg.
`-x`	`--center-x`	X coordinate	Mandatory args. The Map Point corresponding to the image center.
`-y`	`--center-y`	Y coordinate	
`-r`	`--pixel-size`	decimal number	Mandatory arg. Pixel size (expressed in Map Units)
`-w`	`--width`	output image width	Mandatory args. The output image dimensions (expressed in pixels)
`-h`	`--height`	output image height	
`-i`	`--image-type`	the preferred image format to be used for individual tiles	Optional arg, default = JPEG Allowable values are: TIFF, PNG, GIF and JPEG *
`-q`	`--quality`	the adjustable quality setting for *lossy* compression algorithms supporting such an option	Optional arg, default = 75 for JPEG Ignored in any other case.
`-c`	`--transparent-color`	hexadecimal RGB color 0xRRGGBB	Optional arg, default NONE
`-b`	`--background-color`	hexadecimal RGB color 0xRRGGBB	Optional arg, default BLACK (i.e. 0x000000)

* requesting an output GIF image may produce a failure, due to color space limitations.

supporting Grids

a very commonly found Raster GIS data, is represented by **Grid**s: a Grid isn't a digital image at all, but actually is a georeferenced rectangular matrix, and each cell stores some *numeric value* (i.e. *some georeferenced physical measure*). Grids are very often used to store **DEM**s (***D**igital **E**levation **M**odels*): in this case each cell value corresponds to the mean elevation for that map cell.

Transforming some Grid into a corresponding GeoTIFF is a quite trivial task: you simply have to set an appropriate *false-color scale* in order to transform each cell-value into a corresponding pixel.

The `rasterlite_grid` utility tool supports you in an easy way during this Grid to GeoTIFF transformation.

rasterlite_grid **arguments:**

Short format	Long format	Expected value	Notes
`-?`	`--help`		Prints the arg-list and then exits
`-g`	`--grid-path`	the Grid path [input]	Mandatory arg. *
`-c`	`--color-path`	the Color Table path [input]	Mandatory arg. **
`-t`	`--tiff-path`	the GeoTIFF path [output]	Mandatory arg.
`-p`	`--proj4text`	a valid PROJ.4 string	Mandatory arg. ***
`-f`	`--grid-format`	the Grid format	Mandatory arg. * May be one of: • ASCII • FLOAT
`-n`	`--nodata-color`	hexadecimal RGB color 0xRRGGBB	Optional arg, default BLACK (i.e. 0x000000)
`-v`	`--verbose`		Verbose output [single scanline progress]

* supported Grids may have one of the following formats:

- an **ASCII** Grid is a single file [usually identified by the **.asc** suffix]. When using an ASCII Grid you are required to set the complete grid path [*including the suffix*], as in:
  - `-g grid-name.asc`
- a **FLOAT** Grid is shipped as a couple of corresponding files (*sharing a common prefix*):
  - the `grid-prefix.hdr` file will contain the Grid headers infos.
  - the `grid-prefix.flt` file will containe the Grid binary data.
  - both files are required in order to retrieve the Grid data. When using a FLOAT Grid you are required to set the abstract grid path [*omitting any suffix*], as in:
  - `-g grid-name`

** the Color Table simply is a text file. Each row has to contain the following tokens:

a) the minimum range value
b) the maximum range value
c) an hexadecimal RGB color corresponding to this range of values.

Each token has to be separated from the following using any sequence of SPACE or TAB characters.
Each row has to be terminated using an LF or CR+LF

The following is a simple example of well formatted Color Table:

```
-4 -3 0xcddff1
-2 -2 0xcedff1
-1 -1 0xcfdff1
0 0 0xd0e0f0
1 2 0xd0ffd0
3 4 0xcffed0
```

*** a PROJ.4 string represents the complete set of geodetic parameters identifying some SRID.

Examples:

PROJ.4 string	Corresponding EPSG SRID
`-p "+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs"`	**4326** WGS 84
`-p "+proj=utm +zone=32 +ellps=WGS84 +datum=WGS84 +units=m +no_defs"`	**32632** WGS 84 UTM zone 32N

You can easily retrieve such strings by querying the EPSG dataset, i.e. querying the `spatial_ref_sys` table contained in every SpatiaLite's DB.

Using the RasterLite C API

The **RasterLite** library implements a simple, straightforward and really easy-to-use C API.
This means that you can immediately use RasterLite into your own **C** or **C++** code; but it would be really easy to implement appropriate bindings for many other languages, namely **Python**.
And the RasterLite API is splendidly suited to allow an easy and painless integration for **FastCGI** applications.

Opening and closing a RasterLite data source:

```
#include <rasterlite.h>

void * rasterliteOpen(const char *path, const char *table_prefix);

void rasterliteClose(void * handle);
```

First of all you have to call **`rasterliteOpen()`** in order to establish a permanent connection to some Raster Data Source, specifying the DB path and the *table_prefix* identifying the data source.
This function will always return a reference to an **opaque handle** identifying the data source for any subsequent function call. Such handle may well be an invalid one (if some error occurred) so you must carefully check for such an evenience.

Before terminating, your program must absolutely call **`rasterliteClose()`** for each handle obtained by **`rasterliteOpen()`**, in order to disconnect the data source and freeing any related memory allocation.
After calling **`rasterliteClose()`** the handle becomes an invalid one, and any subsequent use of the terminated handle will likely cause a program crash.

Error checking:

```
#include <rasterlite.h>

int rasterliteIsError(void * handle);

const char * rasterliteGetLastError(void * handle);
```

You can check if the previous RasterLite function call raised an error condition calling **`rasterliteIsError()`**; this will return **0** (*no error*) or **1** (*some error was encountered*).

You can get a full error message calling **`rasterliteGetLastError()`**; if no error was set this will return **NULL**

As a general rule, RasterLite's functions can return the following status values:

- **`RASTERLITE_OK`** (*success*)
- **`RASTERLITE_ERROR`** (*some error occurred*) .

Image retrieving:

```
#include <rasterlite.h>

int rasterliteGetRaster (void * handle, double cx, double cy,
                         double pixel_size, int width, int height,
                         int image_type, int quality_factor,
                         void **raster, int *size);

int rasterliteGetRaster2 (void * handle, double cx, double cy,
                          double pixel_x_size, double pixel_y_size,
                          int width, int height, int image_type,
                          int quality_factor, void **raster, int *size);

int rasterliteGetRasterByRect (void * handle, double x1, double y1, double x2,
                               double y2, double pixel_size, int width,
                               int height, int image_type, int quality_factor,
                               void **raster, int *size);

int rasterliteGetRasterByRect2 (void * handle, double x1, double y1, double x2,
                                double y2, double pixel_x_size,
                                double pixel_y_size, int width, int height,
                                int image_type, int quality_factor,
                                void **raster, int *size);
```

Basically, there is simply only one function, i.e. **`rasterliteGetRaster2()`**; the others merely are convenience functions allowing a more flexible way to specify arguments.

You always have to specify the raster dimension, using the **`width`** and **`height`** args.
You can select the requested image format using the **`image_type`** arg, and you can optionally specify a compression factor using the **`quality_factor`** arg (*this will be ignored if not supported by the image format; and when it specifies an invalid value a sound default will be wisely replaced*). You must use one of the following pre-defined constant values for **`image_type`**:

GAIA_JPEG_BLOB
GAIA_GIF_BLOB
GAIA_PNG_BLOB
GAIA_TIFF_BLOB

You can set the center point Map Coordinates using the **`cx`** and **`cy`** args, or alternatively you can set the frame extreme points using the **`x1`**, **`y1`**, **`x2`** and **`y2`** args.
You can set the required pixel size for both axes using the **`pixel_size`** arg, or alternatively you can set a distinct pixel size for each axis using the **`pixel_x_size`** and **`pixel_y_size`** args.

If the function call was a successful one, you can now find the image you've requested stored inside the memory allocation pointed by **`raster`**, and with a **`size`** length. In case or failure (retvalue **0**), then **`raster`** will point to **NULL**, and **`size`** is always **0**.

You are requested to free any memory allocated by the image when you don't need this to be used any longer; so you have to explicitly call: **`free(raster);`**

Exporting a GeoTIFF:

```
#include <rasterlite.h>

RASTERLITE_DECLARE int rasterliteExportGeoTiff (void *handle,
                       const char *img_path, void *raster, int size,
                       double cx, double cy, double pixel_x_size,
                       double pixel_y_size, int width, int height);
```

You can export a GeoTIFF image as well, following a two step process:

- first you have to generate an *in-memory generic* TIFF image, by calling one of the above `rasterliteGetRaster...()` functions.
- and then you have to call `rasterliteExportGeoTiff()` in order to save a true GeoTIFF on the file system, as the following code snippet shows:

```
#include <stdio.h>
#include <rasterlite.h>

  void * handle;
  int srid;
  const char *auth_name;
  int auth_srid;
  const char *ref_sys_name;
  const char *proj4text;
  const char *img_path = "test-geotiff.tif";
  void *raster;
  int size;
  double cx = 11.5;  /* the center point X coord */
  double cy = 43.5;  /* the center point Y coord */
  int dim = 1024;    /* the GeoTIFF width and height */
  double pix_size = 0.5; /* the pixel size */
/* opening the data source */
  handle = rasterliteOpen("mydb.sqlite", "my_rasters");
  if (rasterliteIsError(handle))
  {
    printf("ERROR: %s\n", rasterliteGetLastError(handle));
    rasterliteClose(handle);
    return;
  }
/* querying for Reference System and Extent */
  if (rasterliteGetSrid(handle, &srid, &auth_name, &auth_srid,
     &ref_sys_name, &proj4text) != RASTERLITE_OK)
  {
    printf("ERROR: %s\n", rasterliteGetLastError(handle));
    rasterliteClose(handle);
    return;
  }
  if (rasterliteGetRaster (handle, cx, cy, pix_size, dim, dim, GAIA_TIFF_BLOB, 0,
    &raster, &size) == RASTERLITE_OK)
  {
    if (rasterliteExportGeoTiff (handle, img_path, raster, size, cx, cy,
      pix_size, pix_size, dim, dim) != RASTERLITE_OK)
        fprintf (stderr, "write error on \"%s\"\n", img_path);
  /* freeing the raster image */
     free (raster);
  }
  else
  {
  /* some error occurred */
    fprintf (stderr, "ERROR: %s\n", rasterliteGetLastError (handle));
  }
/* closing the data source */
  rasterliteClose(handle);
```

Reference System and Extent functions:

```
#include <rasterlite.h>

int rasterlingGetSrid(void *handle, int *srid, const char **auth_name,
                      int *auth_srid, const char **ref_sys_name,
                      const char **proj4text);

int rasterliteGetExtent(void *handle, double *min_x, double *min_y,
                        double *max_x, double *max_y);
```

You can use these functions in order to get the Reference System and the Extent for a given data source, as the following code snippet shows:

```
#include <stdio.h>
#include <rasterlite.h>

  void * handle;
  int srid;
  const char *auth_name;
  int auth_srid;
  const char *ref_sys_name;
  const char *proj4text;
  double min_x;
  double min_y;
  double max_x;
  double max_y;
/* opening the data source */
  handle = rasterliteOpen("mydb.sqlite", "my_rasters");
  if (rasterliteIsError(handle))
  {
    printf("ERROR: %s\n", rasterliteGetLastError(handle));
    rasterliteClose(handle);
    return;
  }
/* querying for Reference System and Extent */
  if (rasterliteGetSrid(handle, &srid, &auth_name, &auth_srid,
      &ref_sys_name, &proj4text) != RASTERLITE_OK)
  {
    printf("ERROR: %s\n", rasterliteGetLastError(handle));
    rasterliteClose(handle);
    return;
  }
  if (rasterliteGetExtent(handle, &min_x, &min_y, &max_x, &max_y) !=
    RASTERLITE_OK)
  {
    printf("ERROR: %s\n", rasterliteGetLastError(handle));
    rasterliteClose(handle);
    return;
  }
  printf("'%s'\n", rasterliteGetTablePrefix(handle));
  printf("\tSRID        = %d\n", srid);
  printf("\tAuthority   = %s\n", auth_name);
  printf("\tAuthSRID    = %d\n", auth_srid);
  printf("\tRefSys Name = %s\n", ref_sys_name);
  printf("\tProj4Text   = %s\n", proj4text);
  printf("\tExtent  Min = %1.6f %1.6f\n", min_x, min_y);
  printf("\tExtent  Max = %1.6f %1.6f\n", max_x, max_y);
/* closing the data source */
  rasterliteClose(handle);
```

Miscellaneous functions:

```
#include <rasterlite.h>

const char * rasterliteGetPath(void * handle);

const char * rasterliteGetTablePrefix(void * handle);

const char * rasterliteGetSqliteVersion(void * handle);

const char * rasterliteGetSpatialiteVersion(void * handle);

int rasterliteGetLevels(void *handle);

int rasterliteGetResolution(void *handle, int level, double *pixel_x_size,
                            double *pixel_y_size, int *tile_count);
```

- **`rasterliteGetPath()`**
- **`rasterliteGetTablePrefix()`**
- **`rasterliteGetSqliteVersion()`**
- **`rasterliteGetSpatialiteVersion()`**

These functions are simply intended to give you a little help in checking the main features for a given data source, but are not at all really interesting.

You can use instead the other two function in a more useful way in order to check the Pyramid's Levels available for a data source, as the following code snippet shows:

```
#include <stdio.h>
#include <rasterlite.h>

  void * handle;
  int levels;
  int cur_level;
  double x_size;
  double y_size;
  int tile_count;
/* opening the data source */
  handle = rasterliteOpen("mydb.sqlite", "my_rasters");
  if (rasterliteIsError(handle))
  {
    printf("ERROR: %s\n", rasterliteGetLastError(handle));
    rasterliteClose(handle);
    return;
  }
/* querying the available Pyramid's Levels */
  levels = rasterliteGetLevels(handle);
  printf("the '%s' data source contains %d Pyramid's Levels:\n\n",
    rasterliteGetTablePrefix(handle), levels);
  for (cur_level = 0; cur_level < levels; cur_level++)
  {
    rasterliteGetResolution(handle, cur_level, &x_size, &y_size, &tile_count);
    printf("level %d of %d] x_size=%1.6f y_size=%1.6f tiles=%d\n",
      cur_level + 1, levels, x_size, y_size, tile_count);
  }
/* closing the data source */
  rasterliteClose(handle);
```