Chapter 8
Practical Tile Storage

The previous chapter gave overviews for several different methods for storing tiled images. In this chapter we will present two fully-implemented techniques for storing tiled images together in large files. This type of method proved to be the best performing for writing, random reading, cached reading, and bulk copying. Furthermore, it is rather simple to implement. The first implementation shown is a fully functional method for writing and reading tile files and takes only about 200 lines of Java code for the reading, writing, and indexing methods.

Additionally, we will present the techniques with accompanying methods for creating tile indexes. These storage methods are designed to handle large and small sets of tiled images and are portable and updateable.

8.1 Introduction to Tile Indexes

Our goal is to store many hundreds or thousands of tiled images in a single file. This could be done by simply writing each image sequentially to a file. However, there would be no way to retrieve the images individually. There would be no way to know which tile address corresponded to which image. We could store the tile address before each image in our file, as shown in Figure 8.1. The problem with this method should be obvious. In order to access a specific tiled image, we have to scan the whole file. For tiled images sets of any significant size this method would be prohibitively inefficient. Instead we need to create a separate index into the file that will allow us to quickly look up the location of a specific tile in the file.

There are two principal ways we can construct the index:

- Sequential list of tile address to file position pairs
- Direct lookup table of file positions.

The simplest method is just to store the tile address, the position in the file and the size of the tiled image in a sequential list. This method is shown in Figure 8.2. The sequential list must still be searched for each tile query. However, the index

J.T. Sample and E. Ioup, *Tile-Based Geospatial Information Systems: Principles and Practices*, DOI 10.1007/978-1-4419-7631-4_8,

Record 1	Tile Address (Row, Column, Level)	Tile Image (N bytes)
Record 2	Tile Address (Row, Column, Level)	Tile Image (N bytes)
Record 3	Tile Address (Row, Column, Level)	Tile Image (N bytes)
Record 4	Tile Address (Row, Column, Level)	Tile Image (N bytes)

Fig. 8.1 Tile file with embedded addresses.

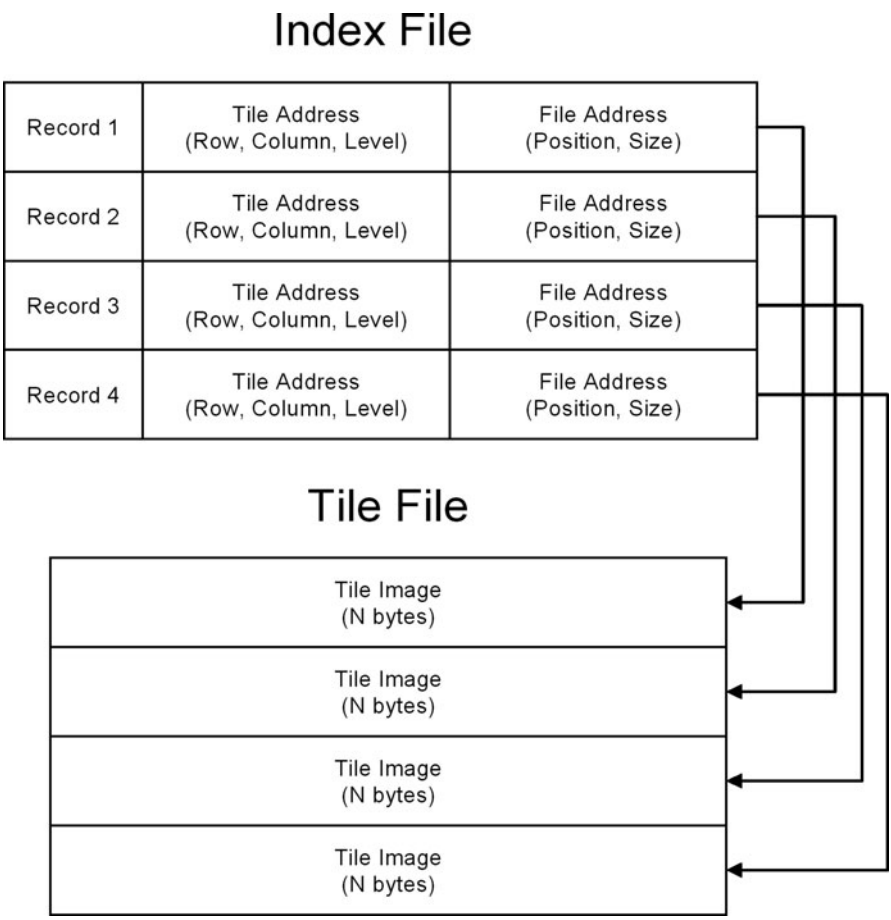

Fig. 8.2 Tile index stored as sequential list of addresses.

information is several orders of magnitude smaller than the actual tiles, thus scanning a separate index file for each query, while still inefficient, is must faster than scanning the whole file. An optimized variant of this method is to sort the data in the tile index. In this fashion a linear search could be used to speed up searching the index list.

The second method is to create a direct lookup table of file positions. For example, as shown in Figure 8.3, tile zoom level 3 has 8 columns and 4 rows, so the lookup table only requires 32 records. The table could be stored in a file in row-major or column-major format. The position of any given record can be directly computed, and only a single seek and read is required to retrieve the file position for a specific tile. We can store a null value in the lookup table to indicate that a tile does not exist for that specific address. If the table is stored in row-major order,

Lookup Table: Zoom Level 3

	Column 0	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7
Row 0	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)
Row 1	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)
Row 2	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)
Row 3	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)	File Address (Position, Size)

Fig. 8.3 Lookup table for zoom level 3.

Equation 8.1 is used to compute the position in the array of addresses.

$$p = j * C + i \tag{8.1}$$

where

$$\begin{aligned} i &= \text{column index} \\ j &= \text{row index} \\ C &= \text{number of columns} \\ p &= \text{position of tile record} \end{aligned}$$

The disadvantage of this approach is that the size of our lookup table file grows by 4 times for each successive level. If the file address is stored as an 8 byte integer

and the size is stored as a 4 byte integer, we need 12 bytes for each record. Zoom level 17 contains 131,072 columns and 65,536 rows for a total of 8,589,934,592 tiles. This would require over 100 gigabytes just for the index file. If we had a tile set with a complete (or nearly complete) coverage of the earth's surface at that resolution, this approach would be appropriate.

However, this is unlikely. Most of the earth's surface is covered with water (liquid and ice) that is rarely imaged at high resolution. Few tile sets will cover even a fraction of the earth's surface. In these cases, we should develop an indexing method that provides direct lookup of tile locations, but also allows us to have lookup tables that cover only a subset of the entire level. This can be easily accomplished by providing for offsets attached to the index table. Rather than having all index tables start at (0,0) and covering the full range of tile addresses, we can provide external start and end addresses for index tables.

The next two sections will each present an algorithm for storing large amounts of tiled images. Each algorithm comes with its own unique method for indexing tiles. Those methods are modified versions of the direct lookup algorithm.

8.2 Storage by Zoom Level

Our first technique for storing tiles is to store all the tiles for a specific zoom level in a single file. This is the same technique that was tested and benchmarked in the previous chapter. This technique uses three files for each zoom level, one file for the tiled images and two files for the index.

The file containing the tiled images is simply a sequential list of tiled images. It first stores a magic number to serve as a sentinel value. Then it stores the tile's address and size. Finally it stores the tiled image data. The sentinel values and tile addresses are stored to make the tile images recoverable in the case that the tile file or index files become corrupted. Figure 8.4 shows the record structure for the tiled image file. Since the tiles do not have to be stored in any particular order, tiles can be written over a period of time. New tiles can be added to the file by simply writing them at the end of the file.

The index storage is slightly more complicated. Recall from the previous section that our lookup table based method can require a very large lookup table for the higher resolution zoom levels. To reduce the required size we have designed a two-step lookup table. We use the same approach to writing the lookup table from Figure 8.3, except that we only store rows in the index file that actually have tiles in them. So if our tile set only has 100 rows, then our tile index will only have 100 rows worth of tile addresses.

To accomplish this we have to create an additional index file, a row index file. This file contains a single value for each row in our set. If the row has any tiles, we store the location of that row's index records from the tile index file. If the row does not have any tiles, we store a null value in the file.

Magic Number	Tile Address (Row, Column)	Tile Size	Tile Image (N bytes)
Magic Number	Tile Address (Row, Column)	Tile Size	Tile Image (N bytes)
Magic Number	Tile Address (Row, Column)	Tile Size	Tile Image (N bytes)
Magic Number	Tile Address (Row, Column)	Tile Size	Tile Image (N bytes)

Fig. 8.4 Structure of tiled image file.

An example of this method is shown in Figure 8.5. We have used the same table from Figure 8.3, but we have assumed that rows 0 and 2 contain zero tiles. In this case, neither of those rows is stored in the index table, and the subsequent table is only half the size. Thus, the advantage of this technique is reduced space requirements for the index file. The disadvantage is that we have to do two seeks and reads to get the tile address. However, as shown in the previous chapter's benchmarks, the performance is still very good.

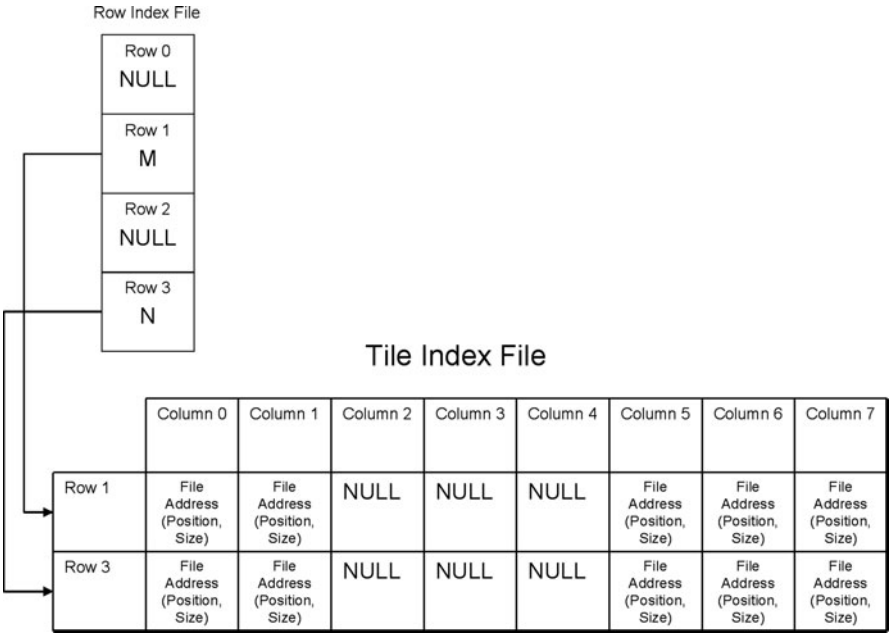

Fig. 8.5 Two-step tile index method.

To get the address for a specific tile, seek to the position of the row pointer in the row index file and read the value. If the value is non-null, use that value to position the tile index file. Then seek additional positions for the column index and read the tile address. Listings 8.1 and 8.2 present example code for writing and reading indexed tiles.

8.3 Introduction to Tile Clusters

The previous method works well and could be modified such that all levels can be contained in a single file. This would require addition of a third index file, a level index file similar to the row index file. Each tile address lookup would require 3 seeks and reads.

However, this method would not address two of the problems discussed in the tile creation chapter. Recall both the performance improvements made possible by caching tiles in memory (Section 6.1) and the requirement to have logically defined sub-groupings of tiles for distributed tile creation (Section 6.3.2). To address both of these requirements we propose a method for grouping tiled images in clusters.

Tiled image layers follow a pyramid type structure, see Figure 8.6. Each level has 4 times the number of tiles as its predecessor. Also, each lower resolution level is based on the image data from the next higher resolution level.

Fig. 8.6 Pyramid structure of tile images.

Our cluster-based grouping method starts by dividing the world into two clusters, (0,0) and (0,1). Figure 8.7 shows that division, and Figure 8.8 shows the structure of a cluster with 5 levels. The tiles that fall into the area marked by address (0,0) are stored in cluster (0,0), and all the tiles that fall into the area marked by address (0,1) are stored in cluster (0,1). By choosing this division we ensure that there are no tiles that overlap both clusters.

The number of tiles for a tile set with l levels is computed with Equation 8.2:

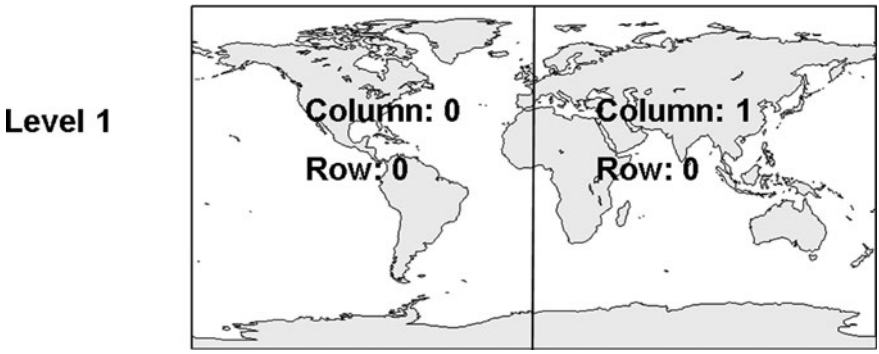

Fig. 8.7 World divided into two clusters.

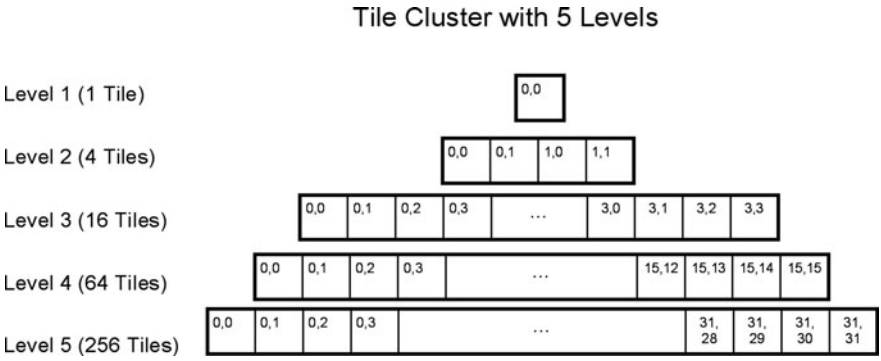

Fig. 8.8 Structure of a cluster with 5 levels.

$$N = \sum_{i=1}^{L} 2^i 2^{i-1} \tag{8.2}$$

The number of tiles for a cluster with l levels is the value from Equation 8.2 divided by two, or as shown in Equation 8.3:

$$N' = \sum_{i=1}^{L} 2^{2i-2} \tag{8.3}$$

8.4 Tile Cluster Files

To store tiles in cluster files, we must first set the number of levels to be stored. For a tile set with a base level of 7, we will need two cluster files, each with 7 levels of tiles and 5,461 tiles. Because the possible number of tiles is fixed for each cluster,

we can build a single fixed length lookup index and store it at the beginning of the cluster file. The index size will be the number of possible tiles times the size of the tile address record. After the index, we can store the tiled images sequentially in the file. Since we have an index, we do not need to store the tiles in any particular order. Figure 8.9 shows the file structure for a cluster file.

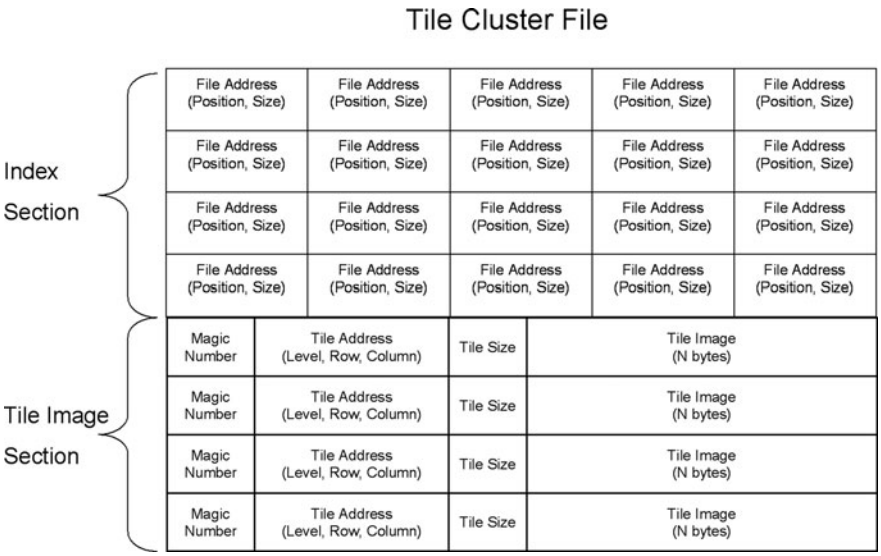

Fig. 8.9 Structure of a tile cluster file.

8.5 Multiple Levels of Clusters

When applying this method to tile sets with several more than 7 levels, we will experience the same problem discussed in Section 8.1. Our index will be too large. Imagine a tile set with only 100 tiles at level 15. Scaled versions of those 100 tiles will give us about 50 additional tiles with levels 14 to 1. That is a total of 150 tiles. If each tile is 50,000 bytes then the size of the tiles in total is 7.5 megabytes. However, a cluster file with 15 levels can have up to 357,913,941 tiles. If each index record takes up 12 bytes, the size of the index table would be 4,294 megabytes, or almost 600 times the size of the actual image data. This is a highly impractical consequence.

To alleviate this problem we allow multiple levels of clusters, with each level covering a continuous sub-range of levels. For example, if we have a tile set with 15 levels, we will have two levels of clusters, one level with contain tile levels 1-7, and the other level of clusters will contain levels 8-15. The first level contains 7 levels,

and the second level contains 8 levels. The indexes for multi-level cluster groups will never grow unmanageably large.

Continuing the example of a multi-level set of clusters, the first set, those with levels 1-7, can only have up to two clusters. While the second set, representing levels 8-15 can contain as many clusters as there are tiles in level 8. This number is 32,768. However, in practice we will only create clusters files when there are tiles that belong in the cluster. Few tile sets will have complete coverage of the whole world at a high resolution, and thus the full 32,768 would never actually be needed. The actual required number would fluctuate based on the size of the tile set.

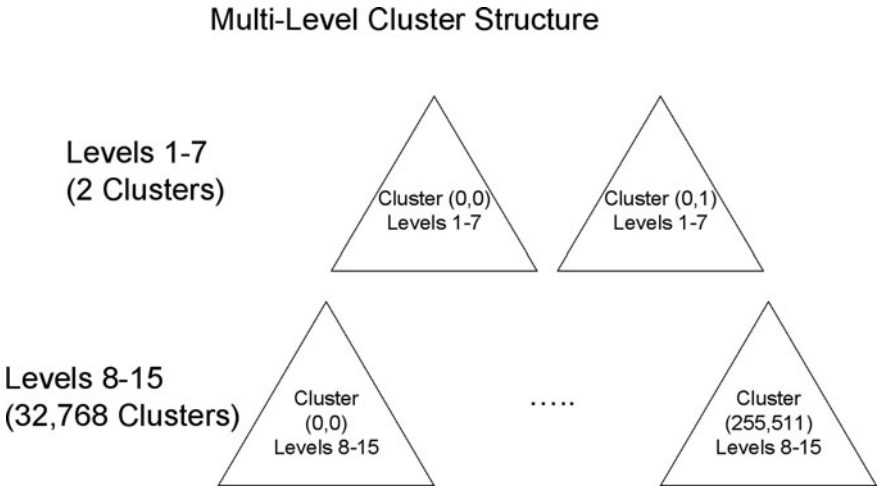

Fig. 8.10 Organization of multiple levels of tile clusters.

8.6 Practical Implementation of Tile Clusters

Listing 8.3 implements a cluster-based tile storage method. Since the internal file structure of our cluster files is relatively simple the implementation is relatively straightforward. The most difficult component of the practical implementation of our cluster-based storage system is the algorithm to determine in which cluster a given tile should be placed. That algorithm can be seen in the methods ”getIndex-Position” and ”getClusterFileTileAddress”.

8.7 Application to Memory Cached Tiles

In Chapter 5, we saw potential performance improvements from holding tiles in memory while they were being created. The cluster-based storage technique works very well with this concept. To implement this with the cluster technique, we first ensure that our clusters are divided small enough to be held uncompressed completely in memory. If that is the case, then we alter our tile creation method to create tiles for one cluster at a time. We modify our clustered tile storage algorithm by simply adding a cache that holds all the tiles in memory as they are written. It writes them to file at the end of the tile creation process. This offers an additional performance improvement. If we write all the tiles at one time, we can write them sequentially and avoid using random file seeks, reads and writes. Random file accesses are generally slower than sequential accesses.

8.8 Application to Distributed Computing

The application of the tile clusters technique to distributed computing should be obvious. Multiple computer systems can be tasked with creating the tiles locally for specific cluster files. The individual cluster files can exist separately and function with minimal interaction, so they are a natural fit for highly distributed computing. After a cluster is completed, the single file can be copied back to a centralized location.

8.9 Performance Optimizations of Tile Cluster Method

There are several other potential performance optimizations available with the clustered storage technique. First, in our example code we opened and closed the various cluster files for each read and write. This is generally slower than maintaining constantly open files and reading and writing from them. Therefore, we might alter our algorithm to keep all the cluster files open throughout the process. However, many systems enforce a limit on the number of open files at any one time. Therefore, to get some performance benefit we can maintain a cache of recently opened files. The cache needs to be of sufficient size to ensure that open files are reused, but it must not be larger than the allowed maximum number of open files.

Since new tiles are written at the end of the file, this technique supports adding tiles over a period of time. When an existing tile is overwritten, the index is updated to point to the new tile. Old tiles remain in the file and take up space. Some developers may want to implement a system to try to re-use that space, either by trying to fit other tile images in the old space or by periodically rebuilding the entire cluster.

Finally, when tiles are served to users from clusters the performance can be quite good. Users typically view tiles for a specific area, and with our system those tiles

would be clustered in the same file. However, there is one case where the performance can be rather poor. Recall our example in which we had tiles in levels 1-15, and separated the clusters into groups of 1-7 and 8-15 levels. If a user is viewing level 8, and requesting several tiles from level 8, the system will have to access a different file for each tile. The benchmarks in the previous chapter showed that using a separate file per tile can be somewhat slow. A workaround to this problem is to build in some overlap in our cluster structure. Instead of a 1-7 and 8-15 break, we will use a 1-8 and 8-15 break. The tiles from level 8 are stored in two places. This does introduce some inefficiency; level 8 can have up to 32,768 tiles. But the read performance improvements may be worth the cost.

Listing 8.1 Output of indexed tiles by zoom level.

```
1
2  public class IndexedTileOutputStream {
3
4          static final long magicNumber = 0x77221lee;
5
6          String imagefilename;
7          String rowindexname;
8          String tileindexname;
9
10         RandomAccessFile imagefile;
11         RandomAccessFile rowindex;
12         RandomAccessFile tileindex;
13
14         long numrows, numcolumns;
15         int rowRecordSize = 8;
16         int tileRecordSize = 8 + 4;
17
18         public IndexedTileOutputStream(String folder, String setname, int level
               ) {
19             imagefilename = folder + "/" + setname + "-" + level + ".tiles";
20             rowindexname = folder + "/" + setname + "-" + level + ".rowindex";
21             tileindexname = folder + "/" + setname + "-" + level + ".tileindex"
                   ;
22             numrows = TileStandards.zoomRows[level];
23             numcolumns = TileStandards.zoomColumns[level];
24
25             try {
26                 imagefile = new RandomAccessFile(imagefilename, "rw");
27
28                 //if the row file is empty, fill it with -1 to indicate empty
                       values
29                 rowindex = new RandomAccessFile(rowindexname, "rw");
30                 if (rowindex.length() == 0) {
31                     rowindex.seek(0);
32                     for (int i = 0; i < numrows; i++) {
33                         rowindex.writeLong(-1L);
34                     }
35                 }
36                 tileindex = new RandomAccessFile(tileindexname, "rw");
37             } catch (Exception e) {
38                 e.printStackTrace();
39             }
40         }
41
42         public void writeTile(long col, long row, byte[] data) {
43
44             try {
45                 //position tile file to write at end of file
46                 long writepos = imagefile.length();
47                 imagefile.seek(writepos);
48
49                 //write tile address and imagedata to file
50                 //write two magic numbers so that tile records can be recovered
                       in case of corrupted file
51                 imagefile.writeLong(magicNumber);
52                 imagefile.writeLong(magicNumber);
53                 imagefile.writeLong(col);
54                 imagefile.writeLong(row);
55                 imagefile.writeInt(data.length);
56                 imagefile.write(data);
57
58                 //update index
59                 updateIndex(col, row, writepos, data.length);
60
61             } catch (IOException e) {
62                 e.printStackTrace();
```

```
63              }
64
65          }
66
67          private void updateIndex(long col, long row, long writepos, int length)
                {
68              try {
69                  //check if row is in the row index
70                  long rowposition = rowRecordSize * row;
71                  rowindex.seek(rowposition);
72                  long rowpointer = rowindex.readLong();
73                  if (rowpointer == -1L) {
74                      //this means the row data is new and not already in the
                            index
75                      rowpointer = tileindex.length();
76                      tileindex.seek(rowpointer);
77                      //write an array of empty values
78                      for (int i = 0; i < numcolumns; i++) {
79                          tileindex.writeLong(-1L);
80                          tileindex.writeInt(-1);
81                      }
82                      //write the position back to the original row index
83                      rowindex.seek(rowposition);
84                      rowindex.writeLong(rowpointer);
85                  }
86                  //compute offset into row for specific col
87                  long offset = rowpointer + col * tileRecordSize;
88                  //position tile index for writing the file address of the tile
                        image
89                  tileindex.seek(offset);
90                  tileindex.writeLong(writepos);
91                  tileindex.writeInt(length);
92              } catch (IOException e) {
93                  e.printStackTrace();
94              }
95          }
96
97          public void close() {
98              try {
99                  imagefile.close();
100                 rowindex.close();
101                 tileindex.close();
102             } catch (Exception e) {
103             }
104         }
105
106     }
```

Listing 8.2 Reading indexed tiles.

```
1
2   public class IndexedTileInputStream {
3
4           String imagefilename;
5           String rowindexname;
6           String tileindexname;
7
8           RandomAccessFile imagefile;
9           RandomAccessFile rowindex;
10          RandomAccessFile tileindex;
11
12          long numrows, numcolumns;
13          int rowRecordSize = 8;
14          int tileRecordSize = 8 + 4;
15
```

```
16          public IndexedTileInputStream(String folder, String setname, int level)
                {
17              imagefilename = folder + "/" + setname + "-" + level + ".tiles";
18              rowindexname = folder + "/" + setname + "-" + level + ".rowindex";
19              tileindexname = folder + "/" + setname + "-" + level + ".tileindex"
                    ;
20              numrows = TileStandards.zoomRows[level];
21              numcolumns = TileStandards.zoomColumns[level];
22
23              try {
24                  imagefile = new RandomAccessFile(imagefilename, "rw");
25                  rowindex = new RandomAccessFile(rowindexname, "rw");
26                  tileindex = new RandomAccessFile(tileindexname, "rw");
27              } catch (Exception e) {
28                  e.printStackTrace();
29              }
30          }
31
32          public byte[] getTile(long col, long row) {
33              try {
34                  //check if row is in the row index
35                  long rowposition = rowRecordSize * row;
36                  rowindex.seek(rowposition);
37                  long rowpointer = rowindex.readLong();
38                  if (rowpointer == -1L) {
39                      //this means the row data is not in the index, and so the
                            tile doesn't exist
40                      return null;
41                  }
42                  //compute offset into row for specific col
43                  long offset = rowpointer + col * tileRecordSize;
44                  //position tile index for reading the position and size of the
                        tile image
45                  tileindex.seek(offset);
46                  long tileposition = tileindex.readLong();
47                  int size = tileindex.readInt();
48                  if (tileposition == -1L) {
49                      //this means that the tile isn't there
50                      return null;
51                  }
52                  //adjust the tile position to skip the magic numbers and
                        address information
53                  long adjustedTilePosition = tileposition + 8 + 8 + 8 + 8 + 4;
54                  byte[] data = new byte[size];
55                  //position the image file and read the image data
56                  imagefile.seek(adjustedTilePosition);
57                  imagefile.readFully(data);
58                  return data;
59              } catch (IOException e) {
60                  e.printStackTrace();
61              }
62              return null;
63          }
64
65      }
```

Listing 8.3 Tile clusters implementation.

```
1
2   public class ClusteredTileStream {
3
4           static final long magicNumber = 0x772211ee;
5           private String location;
6           private String setname;
7           private int numlevels;
8           private int breakpoint;
```

```
9
10          public ClusteredTileStream(String location, String setname, int
                numlevels, int breakpoint) {
11              this.location = location;
12              this.setname = setname;
13              this.numlevels = numlevels;
14              this.breakpoint = breakpoint;
15          }
16
17          public void writeTile(long row, long column, int level, byte[]
                imagedata) {
18
19              //first determine the cluster that will hold the data
20              ClusterAddress ca = getClusterForTileAddress(row, column, level);
21              String clusterFile = getClusterFileForAddress(ca);
22              if (clusterFile == null) {
23                  return;
24              }
25              File f = new File(clusterFile);
26
27              //if the file doesn't exist, set up an empty cluster file
28              if (!f.exists()) {
29                  createNewClusterFile(f, ca.endlevel - ca.startlevel + 1);
30              }
31              try {
32                  RandomAccessFile raf = new RandomAccessFile(f, "rw");
33
34                  //write the tile and info at the end of the tile file
35                  long tilePosition = raf.length();
36                  raf.seek(tilePosition);
37                  raf.writeLong(magicNumber);
38                  raf.writeLong(magicNumber);
39                  raf.writeLong(column);
40                  raf.writeLong(row);
41                  raf.writeInt(imagedata.length);
42                  raf.write(imagedata);
43
44                  //determine the position in the index of the tile address
45                  long indexPosition = getIndexPosition(row, column, level);
46                  raf.seek(indexPosition);
47
48                  //write the tile position and size in the index
49                  raf.writeLong(tilePosition);
50                  raf.writeInt(imagedata.length);
51                  raf.close();
52              } catch (Exception e) {
53                  e.printStackTrace();
54              }
55          }
56
57          public byte[] readTile(long row, long column, int level) {
58              //first determine the cluster that will hold the data
59              ClusterAddress ca = getClusterForTileAddress(row, column, level);
60              String clusterFile = getClusterFileForAddress(ca);
61              if (clusterFile == null) {
62                  return null;
63              }
64              File f = new File(clusterFile);
65
66              try {
67                  RandomAccessFile raf = new RandomAccessFile(f, "r");
68
69                  //determine the position in the index of the tile address
70                  long indexPosition = getIndexPosition(row, column, level);
71                  raf.seek(indexPosition);
72                  long tilePosition = raf.readLong();
73                  int tileSize = raf.readInt();
```

```
74                  if (tilePosition == -1L) {
75                      //tile is not in the cluster
76                      raf.close();
77                      return null;
78                  }
79                  byte[] imageData = new byte[tileSize];
80                  //offset tile position for header information
81                  long tilePositionOffset = tilePosition + 8 + 8 + 8 + 8 + 4;
82                  raf.seek(tilePositionOffset);
83                  raf.readFully(imageData);
84                  raf.close();
85                  return imageData;
86              } catch (Exception e) {
87                  e.printStackTrace();
88              }
89              return null;
90          }
91
92          private long getIndexPosition(long row, long column, int level) {
93              ClusterAddress ca = this.getClusterForTileAddress(row, column,
                    level);
94              //compute the local address, that's the relative address of the
                    tile in the cluster
95              int locallevel = level - ca.startlevel;
96              long localRow = (long) (row - (Math.pow(2, locallevel) * ca.row));
97              long localColumn = (long) (column - (Math.pow(2, locallevel) * ca.
                    column));
98              int numColumnsAtLocallevel = (int) Math.pow(2, locallevel);
99              long indexPosition = this.getCumulativeNumTiles(locallevel - 1) +
                    localRow * numColumnsAtLocallevel + localColumn;
100             //multiply index position times byte size of a tile address
101             indexPosition = indexPosition * (8 + 4);
102             return indexPosition;
103         }
104
105         public ClusterAddress getClusterForTileAddress(long row, long column,
                int level) {
106             if (level > this.numlevels) {
107                 //error, level is outside of ok range
108                 return null;
109             }
110             int targetLevel = 0;
111             int endLevel = 0;
112             if (level < breakpoint) {
113                 //tile goes in one of top two clusters
114                 targetLevel = 1;
115                 endLevel = breakpoint - 1;
116             } else {
117                 //tile goes in bottom cluster
118                 targetLevel = this.breakpoint;
119                 endLevel = this.numlevels;
120             }
121             //compute the difference between the target cluster level and the
                    tile level
122             int powerDiff = level - targetLevel;
123             //level factor is the number of tiles at level "level" for a
                    cluster that starts at "target level"
124             double levelFactor = Math.pow(2, powerDiff);
125             // divide the row and column by the level factor to get the row and
                    column address of the cluster we are using
126             long clusterRow = (int) Math.floor(row / levelFactor);
127             long clusterColumn = (int) Math.floor(column / levelFactor);
128             ClusterAddress ca = new ClusterAddress(clusterRow, clusterColumn,
                    targetLevel, endLevel);
129             return ca;
130         }
131
```

```
132         String getClusterFileForAddress(ClusterAddress ca) {
133             String filename = this.location + "/" + this.setname + "-" + ca.
                    startlevel + "-" + ca.row + "-" + ca.column + ".cluster";
134             return filename;
135         }
136 
137         //this methods create an empty file and fills the index with null
                values
138         void createNewClusterFile(File f, int numlevels) {
139             RandomAccessFile raf;
140             try {
141                 raf = new RandomAccessFile(f, "rw");
142                 raf.seek(0);
143                 long tiles = this.getCumulativeNumTiles(numlevels);
144                 for (long i = 0; i < tiles; i++) {
145                     raf.writeLong(-1L); //NULL position of tile
146                     raf.writeLong(-1L); //NULL size of tile
147                 }
148                 raf.close();
149             } catch (Exception e) {
150                 e.printStackTrace();
151             }
152         }
153 
154         public int getCumulativeNumTiles(int finallevel) {
155             int count = 0;
156             for (int i = 1; i <= finallevel; i++) {
157                 count += (int) (Math.pow(2, 2 * i - 2));
158             }
159             return count;
160         }
161 
162     }
163 
164     public class ClusterAddress {
165 
166         long row;
167         long column;
168         int startlevel;
169         int endlevel;
170 
171         public ClusterAddress(long row, long column, int startlevel, int
                endlevel) {
172             this.row = row;
173             this.column = column;
174             this.startlevel = startlevel;
175             this.endlevel = endlevel;
176         }
177 
178     }
```