# Chapter 7
# Tile Storage

The two previous chapters presented several algorithms for creation of tiled images. Each of those algorithms assumed that some mechanism was in place to support storage and retrieval of tiled images. In this chapter, we will discuss such mechanisms and provide technical guidance on choosing a tile storage system. We will also discuss some advanced topics in tile storage, such as storage of tile metadata and distributed storage of tiles.

## 7.1 Introduction to Tile Storage

Tiled image layers are divided into levels. Each level is divided into rows and columns. Figure 7.1 shows a tiled layer divided into levels, then columns, and then tiles. The general problem of tile storage is linking a tile's address (Layer, Level, Row, and Column) to a binary block of data. That linking should be quickly generated, retrieved, or altered. The practical problem of tile storage is how to organize the blocks of data into levels, rows, and columns so that the tiled images can be efficiently written to and read from disk.

All tiled images are stored in computer files on disk. Tiles can be stored in a separate file for each image, bundled together into larger files, or in database tables. (Database systems use files like any other computer program, so storing tiles in a database indirectly stores them to file.)

The next three sections provide detailed explanations of alternative methods for storing tiles in files. A fourth section provides performance comparisons between the three methods.

J.T. Sample and E. Ioup, *Tile-Based Geospatial Information Systems: Principles and Practices*, DOI 10.1007/978-1-4419-7631-4_7,

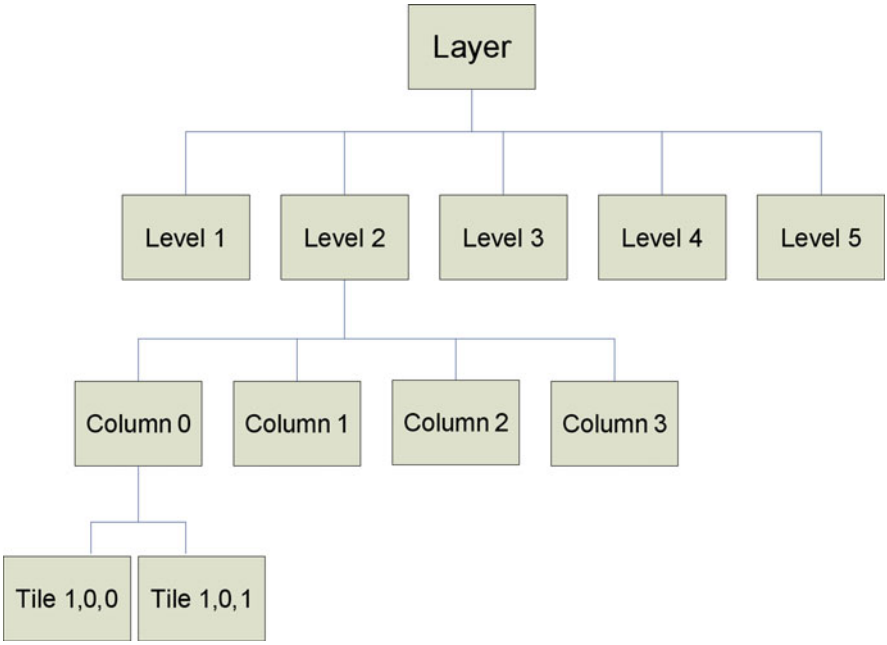


**Fig. 7.1** Tiled image layer divided into components.

## 7.2 Storing Image Tiles as Separate Files

A simple and common method for storing tiled images is to simply store each image in a separate computer file on the computer's file system. Recall from Chapter 5, our tiled images are formatted in standard image formats, like JPEG or PNG. Each of these formats was designed to store an image as a single computer file. Folders (or directories) on the file system can be used to provide structure and organization to the tiled images. For example, we can use a top level folder for the layer, sub-folders within the layer folder for each level, and then subfolders within the level folders for each column. Within the column folders are the individual tiled images for each row in that column. Figure 7.2 shows such an organization.

This type of organization is attractive to developers for several reasons. First, tiles can be addressed directly by simply forming the filename and opening the file. For example, if I want to create a tile, for layer "BlueMarble" at level 7, column 5, and row 4 I can simply create the string "BlueMarble/7/5/4.jpg" and I have the filename for the desired tile. With this method, there is no need for a separate index of tiles. A second benefit is that tiled images can be replaced by newer versions with little impact on the rest of the system.

Finally, and most importantly, building a Web server to host the tiled images in this structure is trivial. Most HTTP servers, including Apache, can, by default, host

files directly on the file system accessible by the sub-path. So, to access a tile over the web, I can construct a URL like the following:
`http://www.sometileserver.com/BlueMarble/7/5/4.jpg`
The HTTP server will simply retrieve the image directly from the file system with minimal configuration.

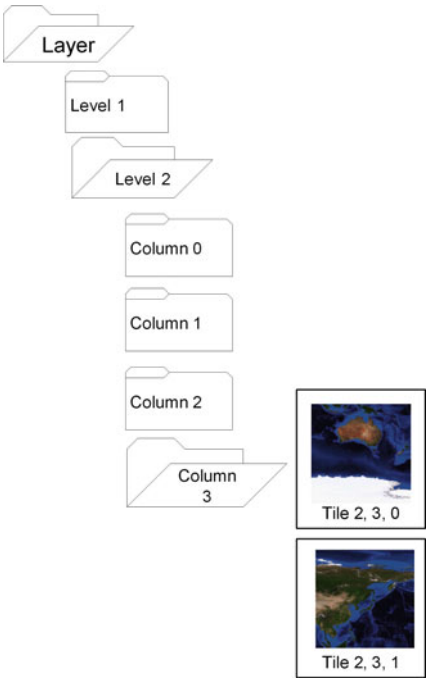


**Fig. 7.2** Folder based organization of tiled images.

However, there are several disadvantages of storing tiles in this method. From the perspective of a software developer, file systems can appear to function by magic. A developer simply names the file he wants, and it appears. He can add to it, delete it, or move it. The system magically knows the size and location of the file, the date it was modified, and which users have what permissions on the file.

In reality, file systems are among the most complicated parts of an operating system. Even though a file can be created with a single line of computer code, there are many things going on behind the scenes that enable that file to magically appear. Space on the hard drive has to be located and allocated for the file. Lists of blocks used to store the file have to be written along with the file's metadata. To store this information, file systems have their own meta-storage allocated. The file system's meta-storage has to be accessed for every file that is created or accessed. In everyday use these operations often seem instant because modern operating systems can cache the file system's meta-storage in memory. However, when writing and reading many

millions of files the memory cache will fail to hold all the needed information, and the file accesses will take much longer. When the small price of a single file access is added to the creation of each and every tile, this method becomes very inefficient and unsuitable for very large tile sets.

Additionally, many file systems do not index files by name. File lookups involve a linear search within a given directory. This is especially problematic given our structure in which a single column folder could hold thousands of image files.

Files are somewhat wasteful with regards to storage space because files are stored in fixed size blocks. A common block size is 4096 bytes, so a file will be broken up into pieces of this size. Files almost always consume an uneven number of blocks. For example, a 10000 byte file will consume three blocks, and a total of 12288 bytes. The average wasted data per file is one half the block size. If the average size of a tiled image file is 50,000 bytes, then the average wasted space is 2048 bytes. Therefore we are wasting around 4% of our storage space with this approach. Four% would be a small price to pay in storage space if this approach yielded significant performance improvements. However, since this approach will likely yield significant performance degradations, the wasted space adds insult to injury.

In many cases tile sets must be copied from one location to another. Perhaps the system that created the tiles is not the same one that will serve them to users over a network, or perhaps multiple systems will be used to serve the same set of tiles. In these cases copies of the entire tile set must be created. To create a copy of the tile set with this storage method requires a separate file access and file write for each tile. This process can take as long as the original tile creation step.

In general, storing tiles as separate image files is a horribly inefficient use of the computer's resources. However, there are a few scenarios in which this is a good approach. First, when dealing with very small tile sets, those with only a few thousand tiled images, this approach is perfectly valid. A more complicated solution would be a waste of time. Second, when the inherent properties of the file system are actually needed, this approach might be useful. For example, a developer might need full use of permissions on each and every tiled image. If the tiles are updated very frequently, and the older tiles can be discarded, this approach might be valid. File systems have sophisticated methods of recapturing used storage space that is no longer needed. Frequent changes to tiles would necessitate this capability.

There is one final scenario in which storing tiles as separate image files makes sense. The File System in Userspace (FUSE) API[1] allows developers to create custom file systems that mimic the properties of a file system on the front end, but store the actual file data with a custom method defined by the developer. A FUSE file system implementation could be created that would allow tiles to be written by software as separate files. On the back end, the tiled images would be stored in an efficient manner that eliminates much of the overhead associated with full featured file systems. This FUSE implementation would also integrate with the HTTP server used to distribute the tiled images. This approach would allow tile system developers

[1] http://fuse.sourceforge.net/

to use a variety of existing, open-source tile creation and distribution tools on very large tile sets.

## 7.3 Database-Based Tile Storage

A second approach to storing tiled images is to store the images within a relational database management system (DBMS) as binary large objects (BLOB). Most modern database systems allow arbitrary size binary arrays to be stored along side structured columns. Using this approach, we can build a ”tiles” table with a column for the image data and other columns for the address components of the tile: level, row, and column. This approach is slightly more complicated than simply storing the data in files. However, since modern database systems use sophisticated techniques for paging of storage this approach might be more efficient. Additionally, we can create indices on the address columns, which could reduce search time.

A disadvantage of this approach is that database systems can be costly in terms of expense, setup, configuration, and maintenance. Like the file system approach, this approach brings a lot of unneeded features that may introduce overhead into the system. Database systems are designed to operate on highly structured data, such as small numeric and character fields. A tile storage system has little need for queries on structured data. Databases also excel at revision control which is unlikely to be needed for a tile system.

As will become apparent in the Comparative Performance section, databases are unlikely to be widely used for storage of tiles. However, there are a couple of scenarios in which they may apply. First, some commercial Web hosting systems provide users with read/write access to a database but not to the file system. If we were forced to use this type of system, we would have to store our tiles in a database. Secondly, if our tile application required sophisticated query functionality we might need a database. For example, if our tiled images also came with extensive metadata like dates, places, names, and keywords that need to be queried for tile retrieval, a database would be useful. A database/file hybrid approach is also a possibility. In this case, the tiles metadata and addresses would be stored in a database, and the image data stored in large flat files.

## 7.4 Custom File Formats

Another approach to storing tiled images is to use a custom designed file format. In this case many tiled images are packed together in a single file instead of in multiple files. This approach necessitates development of an organizational system to keep up with the locations of the tiled images in the single file. It also requires a custom index that allows lookup of tile positions within the large files. This method can offer vastly improved performance, since the inefficiencies of the underlying file

system are mitigated. Another benefit is that the large custom files can more easily be copied from one location to another than many millions of smaller files.

A disadvantage of this system is that, if tiled images change frequently, the custom files may become fragmented. That is, they are littered with out-of-date tiled images that need to be cleaned up. Another disadvantage is that the tiled images cannot be directly accessed by an HTTP server. The server will need a custom module to read the custom formatted files.

In the next chapter we will present two methods for storing images in custom file formats. We will explain the tiled image organization system as well as some high performance indexing schemes.

## 7.5 Comparative Performance

The three previous sections have explained three alternative methods for storing tiled images. In each of those sections we presented some conceptual and practical advantages and disadvantages of each method. In this section we will use some test programs to show the differences in performance.

Benchmarking file writing and reading is very challenging. Modern operating systems perform a lot of caching that can interfere with the results. The best way to measure performance is to create benchmarks that are very close to real-world tasks and run those many times. In this fashion, you can replicate a realistic user environment and average out anomalous results. Before each test we will clear the file system's cache by executing the following Linux command as superuser:

```
echo 3 > /proc/sys/vm/drop_caches
```

This will help ensure each test is performed in a similar environment. The hardware and software configuration for these tests is the same for all tests and is listed in Table 7.1.

| | |
|---|---|
| Operating System | Debian 5 |
| Java Virtual Machine | 1.6.0_15 (64 bit) |
| DBMS | Postgres 8.4, default configuration |
| Processors | 2 2.0Ghz AMD Opteron |
| RAM Size | 16GB DDR2 776Mhz |
| Hard Drive Specification | Dell MD1000 with 15 1TB SAS drives |
| File System | XFS |

**Table 7.1** Test configuration.

### 7.5.1 Writing Tests

This first set of tests will examine writing tiled images. We will write a large number of tile-sized pieces of memory to disk in three different ways and compare the results. In each of the writing tests, we will write tile-sized pieces for each tile in zoom levels 5 through 11. Zoom levels 5 though 11 have 512; 2,048; 8,192; 32,768; 131,072; 524,288; and 2,097,152 tiles, respectively. Each piece of data will be 50,000 bytes in length. The data we write will be simple arrays of random or zero data. We are concerned only with testing the different types of I/O, so the actual contents of the files are not important. We will run each test 30 times to get average performance numbers.

To represent the three methods, we have written three simple implementations. The first implementation writes each tile to a separate file. The second implementation writes all the tiles into a single file for each zoom level and includes an index of tile locations. The third implementation writes all the tiles into a single database table for each zoom level. Each test writes the data to new files and not over existing files.

Listing 7.1 shows the three implementations. In the section "WriteTilesSingleFile" we reference the classes IndexedTileOutputStream and IndexedTileInputStream. These classes are part of the first tile storage implementation discussed in the next chapter and their code is presented there. Table 7.2 shows the results from running the write tests 30 times each. The mean times are in seconds. ¿From this

| Level | Number of Tiles | Single File per Tile | | Single File per Level | | Database Table per Level | |
|---|---|---|---|---|---|---|---|
| | | Mean | StdDev | Mean | StdDev | Mean | StdDev |
| 5 | 512 | 0.1049 | 0.027 | 0.086 | 0.022 | 0.683 | 0.033 |
| 6 | 2,048 | 0.8477 | 0.075 | 0.257 | 0.029 | 2.654 | 0.198 |
| 7 | 8,192 | 3.5807 | 1.623 | 1.090 | 0.115 | 10.540 | 0.509 |
| 8 | 32,768 | 14.2025 | 1.857 | 3.795 | 0.187 | 42.145 | 1.140 |
| 9 | 131,072 | 56.7045 | 2.567 | 21.532 | 0.265 | 167.979 | 3.964 |
| 10 | 524,288 | 244.9717 | 3.862 | 91.684 | 0.695 | 673.950 | 12.783 |
| 11 | 2,097,152 | 999.9249 | 27.582 | 383.365 | 2.762 | 2767.647 | 67.018 |

**Table 7.2** Mean times in seconds and standard deviations from 30 trials of write tests.

table we can see that writing multiple tiles to a single large file yields the best performance. Writing each tile to a separate file takes 2 to 3 times the amount of time. Writing tiles to a DBMS takes 5 to 10 times the amount of time. Figure 7.3 plots the results in terms of average write per tile. The write times for each level are fairly consistent.

Many DBMS systems support bulk imports of data. It would be possible to write tiles out using the fast single file method and then import the data into the database. We have not benchmarked this procedure. Though it would offer some improvement in write performance, it would still be slower than simply writing to the single file. We will see in the next section that reading from the database is also significantly slower.

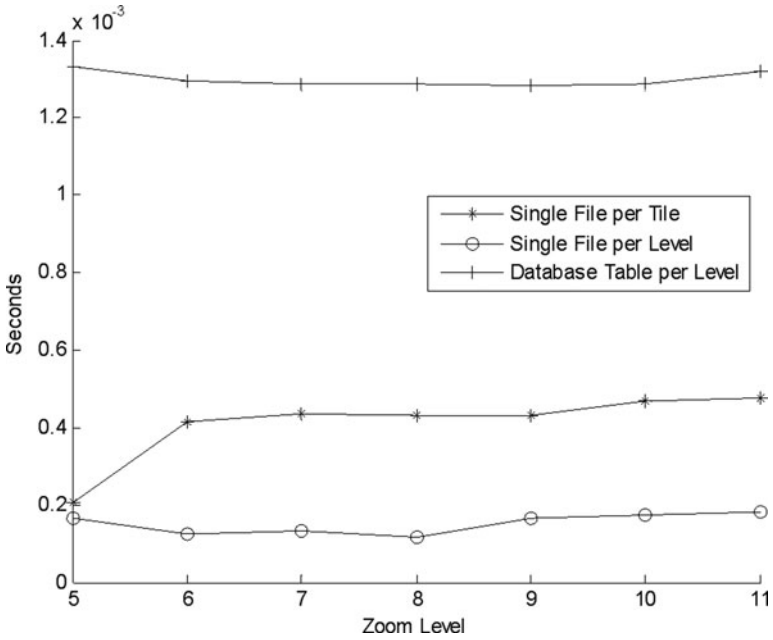


**Fig. 7.3** Plot of average write times per tile.

## *7.5.2 Reading Tests*

For the reading tests, we will use the tiles written in the previous step. The first test will mimic random access of tiles stored on disk, and the second test will mimic random access of tiles cached in memory by the operating system.

### 7.5.2.1 Random Tile Access Tests

For this test we will generate a single random list of tiles of levels 5 through 11. The list will contain 10,000 tile addresses. For each of the three file storage methods, we will iterate over the list of tiles and read each tile from disk. The code for the test is shown in Listing 7.2, and the results are shown in Table 7.3. In this test the single file per level method is fastest, but the database method is a close second. The single file per tile method is slowest.

| | Single File per Tile | Single File per Level | Database Table per Level |
|---|---|---|---|
| Total Read Time (10,000 tiles) | 379.455 seconds | 112.357 seconds | 146.926 seconds |
| Read Time per Tile | 37.9 milliseconds | 11.2 milliseconds | 14.7 milliseconds |

**Table 7.3** Read times for random tile access.

#### 7.5.2.2 Effect of Cached Tile Data

As stated earlier, modern operating systems cache disk file data in memory to speed up access. This test will demonstrate and measure the effect of such caching. In the previous test we read 10,000 random tiles from disk. In this test, we will read 1000 tiles 20 times. The first read will read from disk, and subsequent reads should pull from system memory.

| Trial | Single File per Tile | Single File per Level | Database Table per Level |
|---|---|---|---|
| 1 | 40.994 | 15.952 | 23.838 |
| 2 | 0.881 | 0.190 | 2.328 |
| 3 | 0.828 | 0.183 | 2.357 |
| 4 | 0.162 | 0.211 | 2.339 |
| 5 | 0.162 | 0.137 | 2.284 |
| 6 | 0.159 | 0.129 | 2.269 |
| 7 | 0.117 | 0.121 | 2.280 |
| 8 | 0.116 | 0.121 | 2.298 |
| 9 | 0.117 | 0.197 | 2.273 |
| 10 | 0.117 | 0.121 | 2.285 |
| 11 | 0.127 | 0.116 | 2.200 |
| 12 | 0.101 | 0.112 | 2.174 |
| 13 | 0.101 | 0.110 | 2.195 |
| 14 | 0.099 | 0.105 | 2.171 |
| 15 | 0.098 | 0.121 | 2.178 |
| 16 | 0.100 | 0.105 | 2.249 |
| 17 | 0.098 | 0.112 | 2.226 |
| 18 | 0.100 | 0.105 | 2.200 |
| 19 | 0.098 | 0.106 | 2.242 |
| 20 | 0.100 | 0.111 | 2.228 |

**Table 7.4** Cached tile read times in seconds.

In Table 7.4, we can see that the first read of the 1000 tiles took by far the longest. Table 7.5 shows the results averaged with and without the first trial. We can see that the average times decreased significantly without the first trial.

| | Single File per Tile | Single File per Level | Database Table per Level |
|---|---|---|---|
| Including first trial | 2.2337 | 0.9232 | 3.3307 |
| Excluding first trial | 0.1937 | 0.1323 | 2.2514 |

**Table 7.5** Average read times in seconds with and without first trial.

Table 7.6 compares the cached and non-cached tile read times. The single file per zoom level sees over an 8 to 1 improvement. The database table per zoom level sees over a 6 to 1 improvement. Finally, the single file per tile sees nearly a 20 to 1 improvement. In all cases, the single file per zoom level performs the best overall.

Consideration of memory cached tile files is important. In most cases the tiles from the top zoom levels will be the most commonly accessed, though they are the

| | Single File per Tile | Single File per Level | Database Table per Level |
|---|---|---|---|
| No Caching | 37.9 | 11.2 | 14.7 |
| With Caching | 1.9 | 1.3 | 2.2 |

**Table 7.6** Cached versus non-cached tile read times in milliseconds.

lowest resolution. Tiled map clients will often start with a default view at the world or national level. Users will then zoom in to the specific areas they wish to view. Following this process will cause the top level tiles to be seen by almost all users. A very significant performance improvement can be realized by holding the most commonly accessed tiles in memory, either implicitly by the operating system or explicitly by the tile serving system.

## 7.6 Storage of Tile Metadata

So far we have not considered the need to store metadata about our source images and tiled images. Metadata includes all of the non-imagery data that might be needed. Important pieces of metadata that should be stored along side the tiled images include the date and version number of each tile and the source image(s) used to create each tile. Technical details like the resolution of source imagery used to make the tile or the original map level of the source data should also be included.

A system for maintaining tiled image sets should know which source images have been used to create which tiles so it can perform proper updates to those tiles when the source imagery changes. Large tiled image sets are often created from heterogeneous collections of imagery. Users of a tile-based system will want to know specifically what data was used to create the tiles.

This data is typically smaller than the image data. It can be stored in manners similar to storage of tiled images. In the case where we used a separate file for a tiled image, we could make a separate file for the metadata. We could also put the tile metadata in a database table or packed together in large files with tiled images. The specific means of storing of metadata is not as important as understanding and fulfilling the need to keep up with the data.

## 7.7 Storage of Tiles in Multi-Resolution Image Formats

The two key benefits of a tile-based system are that:

- Tiles are stored pre-rendered, exactly as needed for user consumption.
- Lower resolution views are pre-generated and quickly available.

The primary drawback of tile-based systems is the source imagery must undergo extensive reformatting. Multi-resolution image formats like JPEG2000 and MrSID

are a possible alternative to this reformatting. As is, they meet one of the two key requirements for a tile based system. They use image transforms (typically wavelet based) to generate a multi-resolution encoding of an image. The multi-resolution views can be used to provide the lower resolution zoom level imagery for a tile-based system.

However, these formats do not meet the first requirement. To get a useable sub-image from a wavelet encoded image, several steps must be performed. Because the data is stored in multiple resolutions in multiple places in the file, several file seeks and reads are required. Use of these formats is a tradeoff. They eliminate the need for pre-processing and require less storage space, but they will always require more processing and I/O for tiled image retrieval.

## 7.8 Memory-Cached Tile Storage

In some cases tile retrieval performance must be as fast as possible. This can be a requirement to support real-time applications or to support many millions of users. In these cases, developers may want to create a method for caching entire tile sets in memory. Tile sets will still be archived to file but will be held in memory at run-time. Sizable tile sets will have to be spanned over several computers for this approach to work. Software systems like Memcached[2] are designed for exactly this type of problem. Memcached is used to cache large data sets in the memory of many separate computers.

## 7.9 Online Tile Storage

So far we have considered storing tiles in files on a computer's file system or memory. There are online file storage alternatives. Several services exist which allow Web accessible storage space to be rented. These services provide the storage space hosting with a high degree of reliability, often with multiple backups. One such service is Amazon's Simple Storage Service (S3)[3]. S3 is accessible through a web services interface that allows users to write and read data over HTTP. S3 uses a simple key-value storage system. Data objects (similar to BLOBs) are stored and accessible with a key. The key is used in the formation of an HTTP URL for access to the resource. For example, the following URL could be used to retrieve the binary resource.

`http://www.somestorageservice.com/mykey`

Since tiled images are discretely addressable and designed for use over HTTP, approaches like this are promising for tile-based systems. The primary disadvan-

[2] `http://memcached.org/`

[3] `http://aws.amazon.com/s3/`

tages of this type of storage will be cost and efficiency. However, for very large tile sets with many users, this type of system might be more cost-effective than the required hardware and bandwidth of a self-hosted solution.

**Listing 7.1** Write test implementations.

```
1
2       public static void writeTileMultipleFiles(String outputFolder, int cols,
            int rows) {
3           File f = new File(outputFolder);
4           f.mkdirs();
5           byte[] data = new byte[byteSize];
6           for (int i = 0; i < cols; i++) {
7               String folderName = outputFolder + "/" + i;
8               File folder = new File(folderName);
9               folder.mkdirs();
10              for (int j = 0; j < rows; j++) {
11                  File tileFile = new File(folderName + "/" + j + ".bin");
12                  FileOutputStream fos;
13                  try {
14                      fos = new FileOutputStream(tileFile);
15                      BufferedOutputStream bos = new BufferedOutputStream(fos);
16                      bos.write(data);
17                      bos.close();
18                  } catch (Exception e) {
19                      e.printStackTrace();
20                  }
21              }
22          }
23      }
24
25      public static void writeTilesSingleFile(String outputFolder, int cols, int
            rows, int level) {
26          File f = new File(outputFolder);
27          f.mkdirs();
28          byte[] data = new byte[byteSize];
29          IndexedTileOutputStream ptos = new IndexedTileOutputStream(f.
                getAbsolutePath(), "testing", new TileRange(0, cols - 1, 0, rows -
                1, level));
30          for (int i = 0; i < cols; i++) {
31              for (int j = 0; j < rows; j++) {
32                  ptos.writeTile(i, j, data);
33              }
34          }
35          ptos.close();
36          String s = ptos.getBinFile();
37          IndexedTileInputStream iii = new IndexedTileInputStream(s);
38          iii.close();
39      }
40
41      public static void writeTileDatabase(String tableName, int cols, int rows,
            int level) {
42          try {
43              Connection c = DriverManager.getConnection("jdbc:postgresql://" +"
                    localhost/" + "tiledb", "user", "password");
44              Statement stmt;
45              stmt = c.createStatement();
46              byte[] data = new byte[byteSize];
47              try {
48                  stmt.execute("DROP TABLE " + tableName);
49              } catch (Exception e) {
50              }
51              try {
52                  stmt.execute("CREATE TABLE " + tableName + "(id bigserial
                        PRIMARY KEY ," + "row bigint , " + "col bigint ," + "image
                        bytea )");
53              } catch (Exception e) {
54                  e.printStackTrace();
55                  return;
56              }
57              PreparedStatement ps = c.prepareStatement("INSERT INTO " +
                    tableName + "(row,col,image) VALUES (?,?,?)");
```

```
58              for (int i = 0; i < cols; i++) {
59                  for (int j = 0; j < rows; j++) {
60                      ps.setLong(1, j);
61                      ps.setLong(2, i);
62                      ps.setBytes(3, data);
63                      ps.execute();
64                  }
65              }
66              try {
67                  stmt.execute("CREATE index " + tableName + "_index on " +
                        tableName + " (col,row)");
68              } catch (Exception e) {
69                  e.printStackTrace();
70                  return;
71              }
72          } catch (SQLException e1) {
73              e1.printStackTrace();
74          }
75      }
```

**Listing 7.2** Random read tests.

```
1       private static void readTilesMultipleFiles(String dataLocation, ArrayList<
            String> lines, int trial, int numreads) {
2           int count = 0;
3           for (String s : lines) {
4               if (count == numreads) {
5                   break;
6               }
7               count++;
8               String[] data = s.split(":");
9               String level = data[0];
10              String column = data[1];
11              String row = data[2];
12              String filename = dataLocation + "/" + trial + "_" + level + "/" +
                    column + "/" + row + ".bin";
13              File f = new File(filename);
14              byte[] bytes = new byte[(int) f.length()];
15              try {
16                  FileInputStream fis = new FileInputStream(f);
17                  BufferedInputStream bis = new BufferedInputStream(fis);
18                  DataInputStream dis = new DataInputStream(bis);
19                  dis.readFully(bytes);
20                  dis.close();
21                  if (count % 1000 == 0) {
22                      System.out.println(count + ":" + bytes.length);
23                  }
24              } catch (Exception e) {
25                  e.printStackTrace();
26              }
27          }
28      }
29
30      private static void readTilesSingleFile(String dataLocation, ArrayList<
            String> lines, int trial, int numreads) {
31          IndexedTileInputStream[] streams = new IndexedTileInputStream[12];
32          int count = 0;
33          for (String s : lines) {
34              if (count == numreads) {
35                  break;
36              }
37              count++;
38
39              String[] data = s.split(":");
40              int level = Integer.parseInt(data[0]);
41              long column = Long.parseLong(data[1]);
```

```
42              long row = Long.parseLong(data[2]);
43              if (streams[level] == null) {
44                  long maxCol = TileStandards.zoomColumns[level] - 1;
45                  long maxRow = TileStandards.zoomRows[level] - 1;
46                  streams[level] = new IndexedTileInputStream(dataLocation + "/"
                        + trial + "_" + level, "testing", level);
47
48              }
49              IndexedTileInputStream itis = streams[level];
50              byte[] bytes = itis.getTile(column, row);
51
52          }
53          for (int i = 0; i < streams.length; i++) {
54              if (streams[i] != null) {
55                  streams[i].close();
56              }
57          }
58
59      }
60
61      private static void readTilesDatabase(ArrayList<String> lines, int numreads
            ) {
62          try {
63              Connection c = DriverManager.getConnection("jdbc:postgresql://" +
                    "localhost/" + "tiledb", "username", "password");
64              Statement stmt;
65              stmt = c.createStatement();
66              int count = 0;
67              for (String s : lines) {
68                  if (count == numreads) {
69                      break;
70                  }
71                  count++;
72                  String[] data = s.split(":");
73                  int level = Integer.parseInt(data[0]);
74                  long column = Long.parseLong(data[1]);
75                  long row = Long.parseLong(data[2]);
76                  String tableName = "tiles_" + level;
77                  ResultSet rs = stmt.executeQuery("SELECT image from " +
                        tableName + " where col=" + column + " and row=" + row);
78                  rs.next();
79                  byte[] bytes = rs.getBytes(1);
80              }
81              stmt.close();
82          } catch (SQLException e1) {
83              e1.printStackTrace();
84          }
85
86      }
```