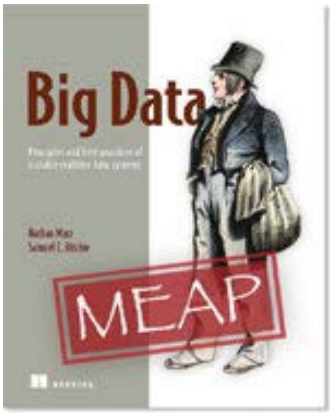
MANNING PUBLICATIONS



Big Data
Principles and Best Practices of Scalable Realtime Data Systems
By Nathan Marz and Samuel E. Ritchie

Pail enables you to vertically partition a dataset and it provides a dead-simple API for common operations like appends, compression, and consolidation. In this article based on chapter 3, author Nathan Marz shows you how Pail makes it easy to satisfy all of the requirements you have for storage on the batch layer.

Data Storage in the Batch Layer with Pail

Pail is a thin abstraction over files and folders from the dfs-datastores library. Pail makes it significantly easier to manage a collection of records in a batch processing context. The Pail abstraction frees you from having to think about file formats and greatly reduces the complexity of the storage code. It enables you to vertically partition a dataset and it provides a dead-simple API for common operations like appends, compression, and consolidation. Pail is just a Java library and underneath it uses the standard file APIs provided by Hadoop. Pail makes it easy to satisfy all of the requirements you have for storage on the batch layer.

You treat a pail like an unordered collection of records. Internally, those records are stored across files that can be nested into arbitrarily deep subdirectories, as shown in figure 1.

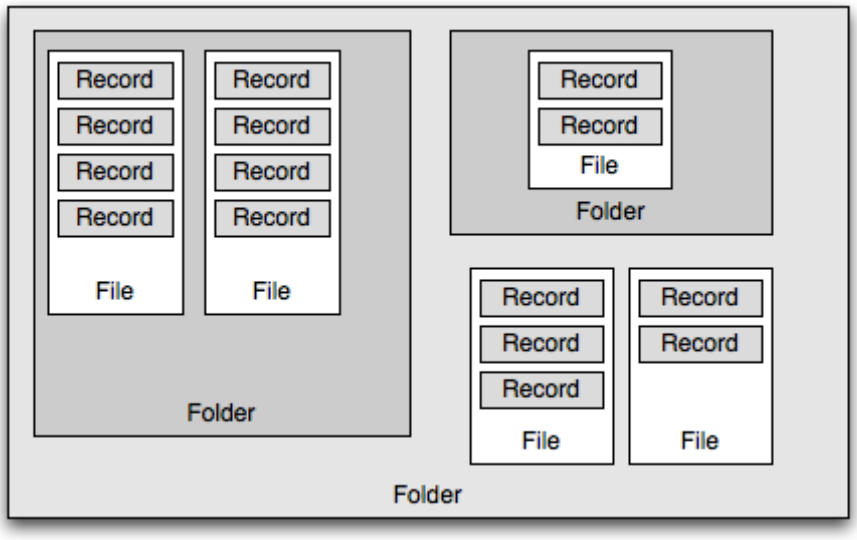


Figure 1 How records are stored in a Pail

Let's dive right into the code to see how Pail works. We'll start off looking at how to do basic operations with Pail, like add records and specify how to serialize and deserialize objects of any type. Then, we'll look at how Pail makes doing mass record operations like appends and consolidations dead-simple. Then, we'll look at how to use Pail to vertically partition your dataset automatically and, finally, you'll see how you can tune the file format Pail uses to enable things like compression.

Basic Pail operations

The best way to understand how Pail works is to follow along and run these commands on your local computer. Your local filesystem will be treated as Hadoop Distributed File System (HDFS) in the example, so you'll be able to see the results of these commands by inspecting the relevant directories on your filesystem. Let's start off by creating a new pail at "/tmp/mypail":

```
Pail pail = Pail.create("/tmp/mypail");
```

If you look on your filesystem, you'll see that a folder for "/tmp/mypail" was created and a file called "pail.meta" exists inside. Ignore that file for now as we'll come back to it later.

As you can see, `Pail.create` returned a Pail object that you can manipulate. Let's use that object to add some records to the pail. This pail stores raw binary data. The following code shows how to write some byte arrays to it.

```
TypedRecordOutputStream os = pail.openWrite();
os.writeObject(new byte[] {1, 2, 3});
os.writeObject(new byte[] {1, 2, 3, 4});
os.writeObject(new byte[] {1, 2, 3, 4, 5});
os.close();
```

If you look inside /tmp/mypail, you'll see a new file inside that contains the records. The file is created atomically, so all the records you created will appear at once—that is, the reader of the pail will not see the file until the writer closes it.

Since files are named using globally unique names, a pail can be written to concurrently by multiple writers without conflicts. Additionally, a reader can read from a pail while it's being written to without having to worry about half-written files.

If a pail already exists and you want to open it, you just use Pail's constructor to get an instance of it. Remember, Pail is backed by a distributed filesystem so multiple processes can be reading/writing to the same pail. Here's an example of opening an existing pail:

```
Pail pail_same = new Pail("/tmp/mypail");
```

If you ran this code on a directory that isn't part of a pail, you would get an exception.

Typed pails

You don't have to work with binary records when using Pail. Pail lets you work with real objects rather than binary records. At the file level, data is stored as a sequence of bytes. To work with real objects, you provide Pail with information about what type your records will be and how to serialize and deserialize objects of that type to and from binary data. You provide this information by implementing the PailStructure interface. Here's how to define the PailStructure for a pail that stores integers:

Listing 1 Defining structure for pail that stores integers

```
public class IntegerPailStructure implements PailStructure<Integer> {
    public Integer deserialize(byte[] bytes) {
        try {
            return new DataInputStream(
                new ByteArrayInputStream(bytes)).readInt();
        } catch (IOException e) {
           throw new RuntimeException(e);
        }
    }

    public byte[] serialize(Integer t) {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(os);
        try {
            dos.writeInt(t);
```

```
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
            return os.toByteArray();
        }

        public Class getType() {
            return Integer.class;
        }

        public List<String> getTarget(Integer t) {
            return Collections.EMPTY_LIST;
        }

        public boolean isValidTarget(String... strings) {
            return true;
        }
    }
```

To create an integer pail, you simply pass that PailStructure in as an extra argument on `Pail.create`:

```
Pail<Integer> intpail = Pail.create("/tmp/intpail",
  new IntegerPailStructure());
```

Now, when writing records to the pail, you can give it integer objects directly and the Pail will handle the serialization. This is shown in the following code:

```
TypedRecordOutputStream int_os = intpail.openWrite();
int_os.writeObject(1);
int_os.writeObject(2);
int_os.writeObject(3);
int_os.close();
```

Likewise, when you read from the pail, the pail will deserialize records for you.

Here's how you can iterate through all the objects in the integer pail you just wrote to:

```
for(Integer record: intpail) {
    System.out.println(record);
}
```

This code will print out integers, just as you expect.

Appends

Pail has a number of common operations built into it as methods. These operations are where you really start to see the benefits of managing your records with Pail rather than managing files yourself. These operations are all implemented using MapReduce so they scale to however much data is in your pail, whether gigabytes or terabytes. The operations are automatically distributed across a cluster of worker machines.

One of Pail's operations is the append operation. Using the append operation, you can add all the records from one pail into another pail. Here's an example of appending a pail called "source" into a pail called "target":

```
Pail source = new Pail("/tmp/source");
Pail target = new Pail("/tmp/target");
target.copyAppend(source);
```

As you can see, the code is super simple. It automates all the stuff you had to do when using files and folders directly. Pail lets you focus on what you want to do with your data rather than worry about how to manipulate files appropriately.

Unlike the examples so far, the append operation won't complete instantly because it's launching an entire MapReduce job. It will block until the operation is complete, and it will throw an exception if for some reason the job failed.

The append operation is smart. It checks the pails to make sure it's valid to append the pails together. So for example, it won't let you append a pail that stores strings into a pail that stores integers.

There's a few kinds of appends you can use. Figure 2 compares these different appends. The append we used in our example was the `copyAppend`, which performed the append by copying all the records from the source pail into the target pail. The `copyAppend` does not modify the source pail and can even be used to copy a pail between two separate distributed filesystems.

	copyAppend	moveAppend	absorb
Cross-filesystem?	Yes	No	Yes
Can append different formats?	Yes	No	Yes
Modifies source pail?	No	Yes	Maybe
Uses filesystem move operations for better performance?	No	Yes	Maybe

Figure 2 Comparing different kinds of appends

Another append is the `moveAppend`. The `moveAppend` works by doing filesystem move operations to move the source pail's files into the target pail. The `moveAppend` operation is much faster than a `copyAppend`, but it has some restrictions. You can't `moveAppend` between two pails if they're stored on different filesystems or are using different file formats to store the data. If you just want to append the contents of a pail into another pail in the most efficient way possible, and don't care if the source pail is modified, use the `absorb` operation. `Absorb` will do a `moveAppend` if possible; otherwise, it will fall back on a `copyAppend`.

Consolidation

Sometimes your records end up spread across lots of small files. This has a major performance cost associated with it when you want to process that data in a MapReduce job since MapReduce will need to launch a lot more tasks.

The solution is to combine those small files into larger files so that more data can be processed in a single task. Pail supports this directly by exposing a `consolidate` method. This method launches a MapReduce job that combines files into larger files in a scalable way. Here's how you would consolidate a pail:

```
pail.consolidate();
```

By default, Pail combines files to create new files that are as close to 128 MB as possible. You can configure the target filesize by passing the filesize in as an argument to consolidate. 128 MB is a good default because it is a typical block size used in HDFS installations.

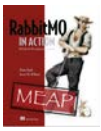
Summary

It's important to be able to think about and manipulate your data at the record level and not at the file level. By abstracting away file formats and directory structure into the Pail abstraction, you're able to do exactly that. The Pail abstraction frees you from having to think about the details of the data storage while making it easy to do robust, enforced vertical partitioning as well as common operations like appends and consolidation. Without the Pail abstraction, these basic tasks are manual and difficult.

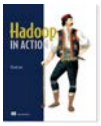
The Pail abstraction plays an important role in making robust batch workflows. However, it ultimately takes up very few lines of code in the code for a workflow. Vertical partitioning happens automatically, and tasks like appends and consolidation are just one-liners. This means you can focus on how you want to process your records rather than the details of how to store those records.

Here are some other Manning titles you might be interested in:

MongoDB in Action
Kyle Banker



RabbitMQ in Action
Alvaro Videla and Jason J.W. Williams



Hadoop in Action
Chuck Lam

Last updated: January 12, 2012