Bloc: Binary Large Objects with Concurrency

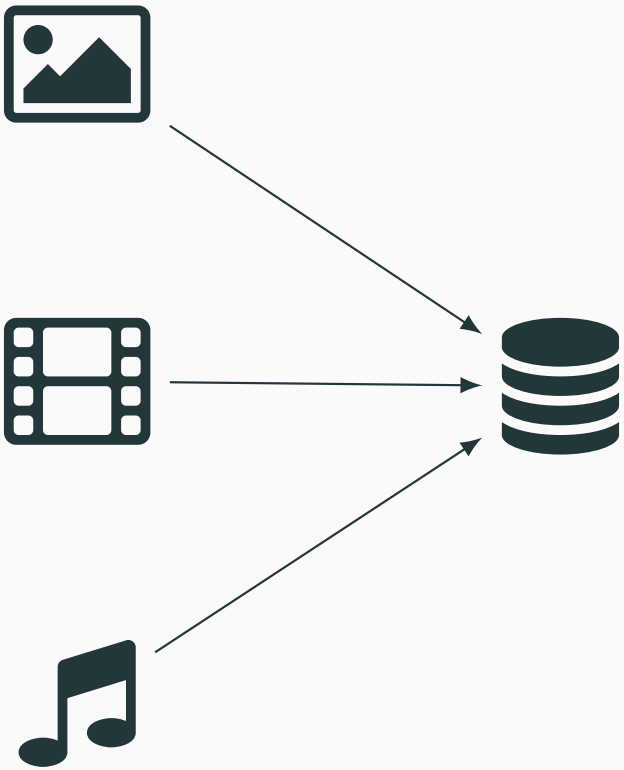
Anshu Avinash
Thesis supervisor: Piyush Kurur

June 17, 2015

Department of CSE, IIT Kanpur

What is a blob?

BLOB stands for “**B**inary **L**arge **OB**ject”.



How to store blobs? I

Store the entire content of the blob in the database.

- PostgreSQL breaks large objects into “chunks” and these chunks are stored in rows in the database.
- MongoDB also divides large objects using the GridFS specification and stores them in the “chunks” collection.

How to store blobs? II

Store the blob in a file and store the file name in the database.

- Number and size of blobs are now limited only by the file system.
- Requires to develop an interface to keep track of all the blobs of a database.

Here comes Bloc!

We provide a library which will keep track of all the blobs stored in a database.

Design

- Our design is inspired from the maildir format[1].

[1] Daniel J Bernstein. *Using maildir format*. 1995.

Design

- Our design is inspired from the maildir format[1].
- All the blobs of a database are stored inside a directory which we also call a BlobStore.

[1] Daniel J Bernstein. *Using maildir format*. 1995.

Design

- Our design is inspired from the maildir format[1].
- All the blobs of a database are stored inside a directory which we also call a BlobStore.
- Each blob is stored as a different file inside the BlobStore.

[1] Daniel J Bernstein. *Using maildir format*. 1995.

Design

- Our design is inspired from the maildir format[1].
- All the blobs of a database are stored inside a directory which we also call a BlobStore.
- Each blob is stored as a different file inside the BlobStore.
- We provide an incremental interface for writing to a blob as well as reading from a blob.

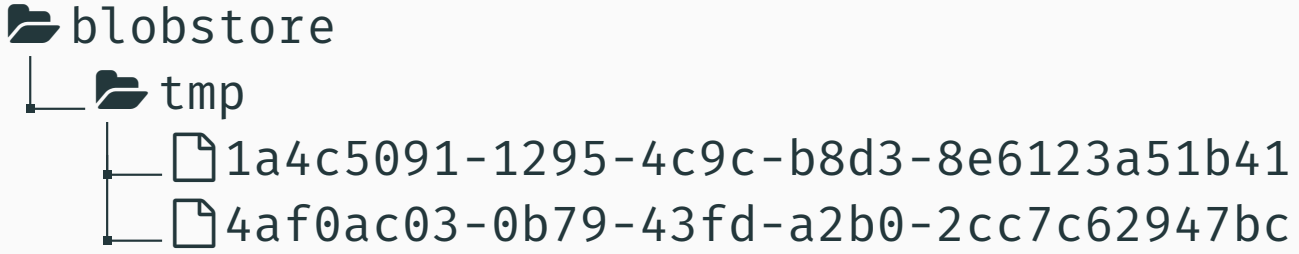
[1] Daniel J Bernstein. *Using maildir format*. 1995.

Creating a new blob

```
openBlobStore :: FilePath -> IO BlobStore
newBlob       :: BlobStore -> IO WriteContext
```

Creating a new blob

```
openBlobStore :: FilePath -> IO BlobStore
newBlob       :: BlobStore -> IO WriteContext
```

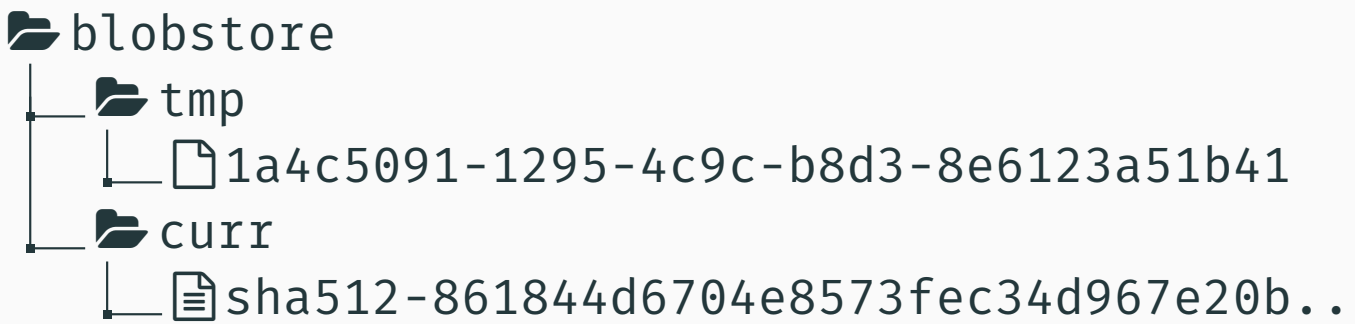


Adding contents to a blob

```
writePartial :: WriteContext
             -> Blob
             -> IO WriteContext
endWrite     :: WriteContext -> IO BlobId
```

Adding contents to a blob

```
writePartial :: WriteContext
             -> Blob
             -> IO WriteContext
endWrite     :: WriteContext -> IO BlobId
```



Reading from a blob

```
startRead    :: BlobId -> IO ReadContext
readPartial  :: ReadContext -> Int -> IO Blob
skipBytes    :: ReadContext -> Integer -> IO ()
endRead      :: ReadContext -> IO ()
```

Deleting blobs

- A blob can be shared by multiple “records” of the database.

Deleting blobs

- A blob can be shared by multiple “records” of the database.
- We provide an interface for garbage collection of deleted blobs.

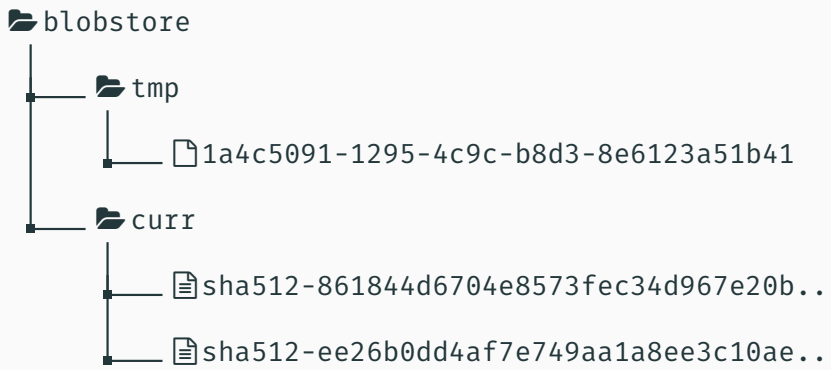
Starting a GC

```haskell
startGC :: BlobStore -> IO ()
```

Starting a GC

```haskell
startGC :: BlobStore -> IO ()
```

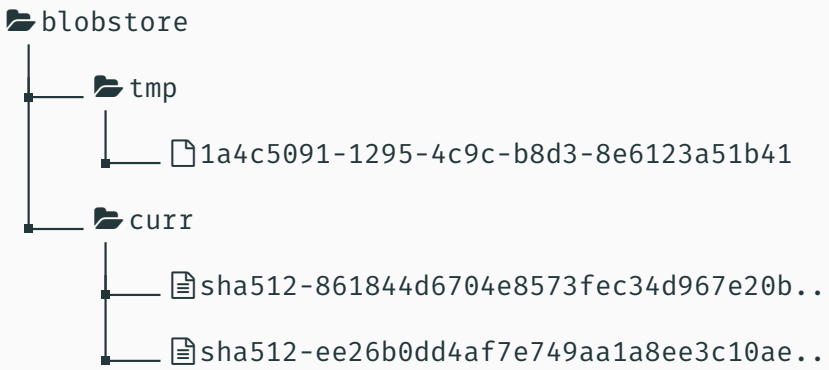
Before startGC



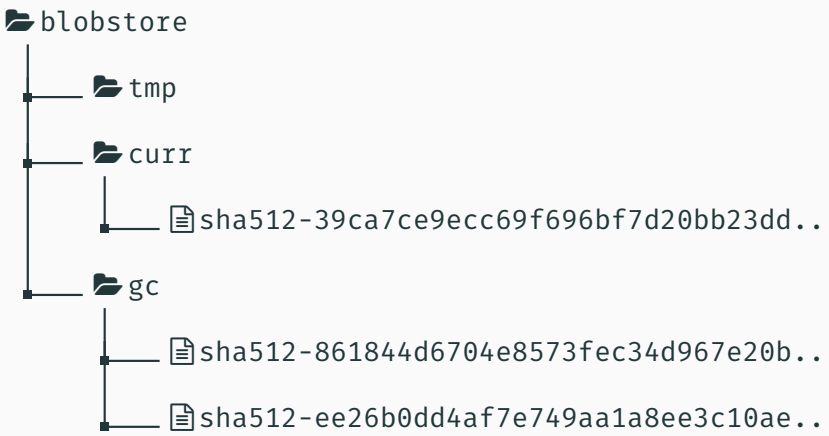
Starting a GC

```
startGC :: BlobStore -> IO ()
```

Before startGC



After startGC



Marking a blob as accessible

```haskell
markAsAccessible :: BlobId -> IO ()
```

Marking a blob as accessible

```haskell
markAsAccessible :: BlobId -> IO ()
```

Before

Marking a blob as accessible

```
markAsAccessible :: BlobId -> IO ()
```

Before

After

Ending a GC

```
endGC :: BlobStore -> IO ()
```

Ending a GC

```haskell
endGC :: BlobStore -> IO ()
```

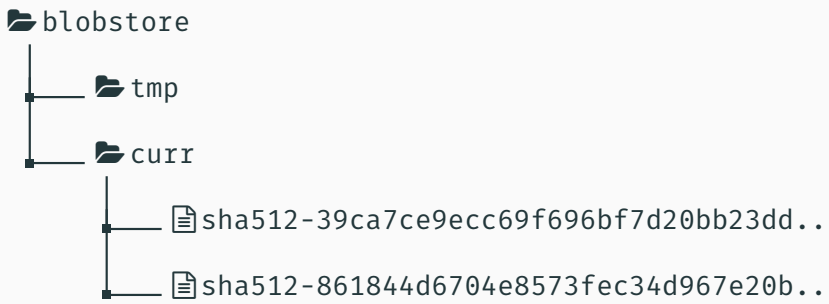
Before

Ending a GC

```
endGC :: BlobStore -> IO ()
```

Before

After



How does this design achieve concurrency?

How does this design achieve concurrency?

Using atomic file operations.

How does this design achieve concurrency?

Using atomic file operations.

- rename

How does this design achieve concurrency?

Using atomic file operations.

- rename
- mkdir

How does this design achieve concurrency?

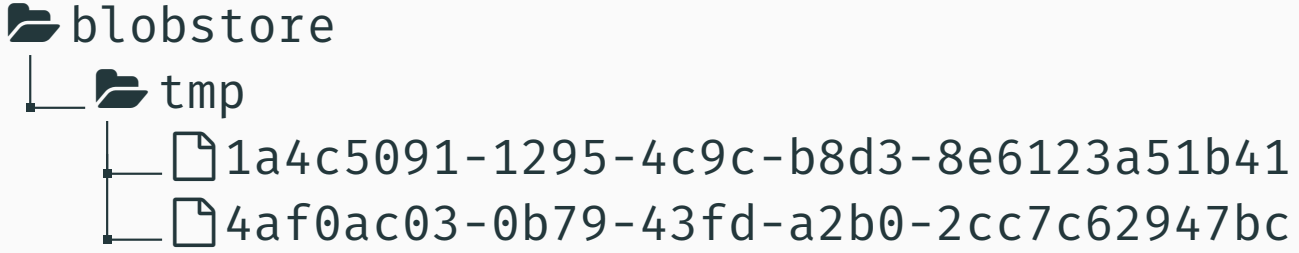
Using atomic file operations.

- rename
- mkdir

Let us revisit our design.

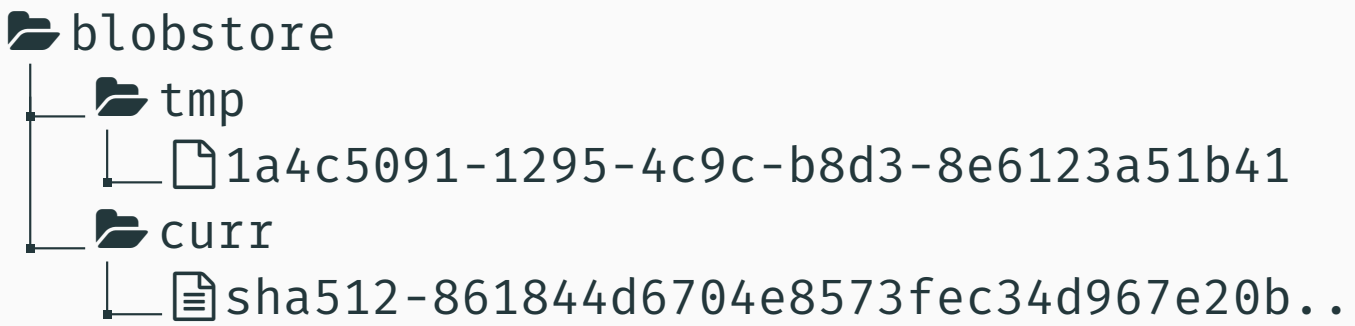
Creating a new blob (Part 2)

```
openBlobStore :: FilePath -> IO BlobStore
newBlob       :: BlobStore -> IO WriteContext
```



Adding contents to a blob (Part 2)

```
writePartial :: WriteContext
             -> Blob
             -> IO WriteContext
endWrite     :: WriteContext -> IO BlobId
```

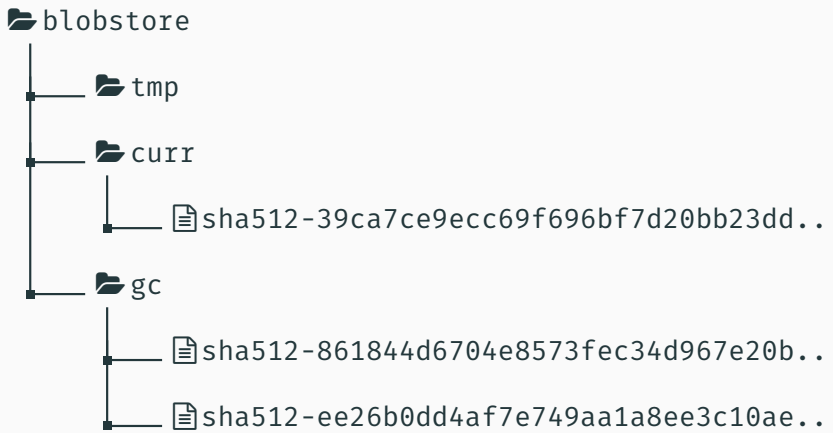


Reading from a blob (Part 2)

```
startRead   :: BlobId -> IO ReadContext
readPartial :: ReadContext -> Int -> IO Blob
skipBytes   :: ReadContext -> Integer -> IO ()
endRead     :: ReadContext -> IO ()
```

GC (Part 2)

```
startGC           :: BlobStore -> IO ()
markAsAccessible  :: BlobId -> IO ()
endGC             :: BlobStore -> IO ()
```



Some helper functions

```
createBlob :: BlobStore -> Blob -> IO BlobId
```

Some helper functions

```
createBlob :: BlobStore -> Blob -> IO BlobId

readBlob :: BlobId -> IO Blob
```

Some helper functions

```haskell
createBlob :: BlobStore -> Blob -> IO BlobId

readBlob :: BlobId -> IO Blob

markAccessibleBlobs :: BlobStore -> [BlobId] -> IO ()
```

Some helper functions

```haskell
createBlob :: BlobStore -> Blob -> IO BlobId

readBlob :: BlobId -> IO Blob

markAccessibleBlobs :: BlobStore -> [BlobId] -> IO ()

recoverFromGC :: FilePath -> IO ()
```

Providing durability

- fsync a file

Providing durability

- fsync a file
- fsync a directory

Future Work

- Proving the correctness of our design

Future Work

- Proving the correctness of our design
- Writing a key-value store

Future Work

- Proving the correctness of our design
- Writing a key-value store
- Supporting bloc on non-POSIX systems